TLP **10** (4–6): 659–674, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000347

*Test case generation for object-oriented imperative languages in CLP**

MIGUEL GÓMEZ-ZAMALLOA, ELVIRA ALBERT
DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain
and GERMÁN PUEBLA
DLSIIS, Technical University of Madrid (UPM), E-28660 Boadilla del Monte, Madrid, Spain

submitted 8 February 2010; revised 11 April 2010; accepted 21 May 2010

Abstract

Testing is a vital part of the software development process. Test Case Generation (TCG) is the process of automatically generating a collection of test-cases which are applied to a system under test. White-box TCG is usually performed by means of *symbolic execution*, i.e., instead of executing the program on normal values (e.g., numbers), the program is executed on symbolic values representing arbitrary values. When dealing with an object-oriented (OO) imperative language, symbolic execution becomes challenging as, among other things, it must be able to backtrack, complex heap-allocated data structures should be created during the TCG process and features like inheritance, virtual invocations and exceptions have to be taken into account. Due to its inherent symbolic execution mechanism, we pursue in this paper that *Constraint Logic Programming* (CLP) has a promising application field in TCG. We will support our claim by developing a fully CLP-based framework to TCG of an OO imperative language, and by assessing it on a corresponding implementation on a set of challenging Java programs.

KEYWORDS: test case generation, symbolic execution, constraint logic programming

1 Introduction

Test Case Generation (TCG) is the process of automatically generating a collection of test-cases which are applied to a system under test. The generated cases must ensure a certain *coverage criterion* (see e.g., Zhu *et al.* 1997 for a survey) which are heuristics that estimates 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; Albert *et al.* 2009). Among all possible forms of TCG, we focus on *static* (i.e., no knowledge about the input data is assumed) and *white-box* TCG (i.e., the program is used for guiding the TCG process). The standard way of performing

* 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 TIN2008-04473-E (Acción Especial) 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.

```
class SLNode {
   int data;
   SLNode next;
}

class SortedList {
   SLNode first;

   public void merge(SortedList l){
      SLNode p1,p2,curr;
      p1 = first; p2 = (1) l.first;
      if ((2) p1.data <= (3) p2.data)// if1
         p1 = (4) p1.next;
      else {
         first = p2; p2 = p2.next; }
      curr = first; //preloop
```

```
// loop
  while ((p1 != null) && (p2 != null)){
  // loopcond1, loopcond2 and loopbody1
      if (p1.data <= p2.data){// if2
         curr.next = p1;
         p1 = p1.next;
      }
      else {
         curr.next = p2;
         p2 = p2.next;
      }
      curr = curr.next;// loopbody2
  }
  if (p1 == null) curr.next = p2;// if3
  else curr.next = p1;
}
```

Fig. 1. Working example: Java source code.

static white-box TCG is program *symbolic execution* (SymEx) (King 1976; Gotlieb *et al.* 2000; Meudec 2001; Müller *et al.* 2004; Tillmann and de Halleux 2008), whereby instead of on actual values, programs are executed on symbolic values, sometimes represented as *constraint variables*. Such constraints are accumulated into *path constraints* as each path of the execution tree is expanded. The path constraints in feasible paths provide pre-conditions on the input data which guarantee that the corresponding path will be executed at run-time.

In this paper, we pursue that *Constraint Logic Programming* (CLP) has a promising application field in TCG, since it inherently combines the use of constraint solvers into its SymEx mechanism. Our main goal is to formalize a whole TCG framework for a realistic object-oriented (OO) imperative language by means of CLP. Our approach consists of two basic parts: first, the imperative program is compiled into an equivalent CLP program and, second, TCG is performed on the CLP program by relying only on CLP's evaluation mechanisms. The main challenges in TCG when dealing with an OO imperative language are related to the creation of complex heap-allocated data structures during the TCG process, and to handling OO features like inheritance and virtual invocations, and exceptions. Besides, when dealing with objects, one needs to take into account all possible *aliasing* among them, since this might affect directly the coverage of the test-cases. Previous approaches strive to define novel specific constraint operators to carry out these tasks (see e.g. Charreteur *et al.* 2009; Schrijvers *et al.* 2009). Instead, in our approach, the whole TCG process is formulated using CLP only, and without the need of defining specific operators to handle the different features. This, on one hand, has the advantage of providing a clean and uniform formalization. And, more importantly, since SymEx is performed on an equivalent CLP program, we can often obtain the desired degree of coverage by using existing evaluation strategies on the CLP side. This gives us flexibility and parametricity w.r.t. the adequacy criteria.

Our approach has been integrated in PET (Albert *et al.* 2010), a Partial-Evaluation based TCG tool, extending its applicability towards real-life OO applications.

2 A CLP-executable object-oriented imperative language

In this section, we define the (CLP) syntax and semantics of the OO imperative language on which our TCG approach is developed, which we call *CLP-decompiled*

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

language. Its main characteristic is that it keeps all features of the original OO language but it is *CLP-executable*, i.e., it can be executed using the evaluation mechanism of CLP languages. When the source imperative language is low-level as bytecode, we use the term CLP-*de*compiled language. In previous work, it has been shown that Java bytecode (and hence Java) can be decompiled into a similar language (Gómez-Zamalloa *et al.* 2009) by relying on the interpretive approach (Futamura 1971) to compilation, proposed in the first Futamura projection. In this approach, the CLP-(de)compilation is obtained by partially evaluating an interpreter for the OO language written in CLP.

Example 1

Figure 1 shows the source code of our running example which implements a merge algorithm on sorted singly-linked lists. Figure 2 shows the CLP-decompiled program automatically generated by our system from the bytecode obtained by compiling the Java program, with some simplifications to improve readability. The correspondence between blocks of the original program and clauses in the decompiled one is shown in comments in the Java code. The main features that can be observed from the decompilation are: (1) All clauses contain input and output arguments and heaps, and an exceptional flag. As in the bytecode, input arguments of non-static methods include the reference *this* (named r(Th)). Reference variables are of the form r(V) and we use the same variable name V as in the program. (2) Java exceptions are made explicit in the decompiled program. Observe predicates `nullcheck`*x*, which capture the exceptions that can be thrown at program points annotated as ⓧ. (3) Conditional statements in the source program are transformed to guarded rules in the CLP one (e.g., if1). (4) Iteration in the source program is transformed into recursion in the CLP program. E.g, the while loop corresponds to the recursive predicate loop.

2.1 Syntax of CLP-decompiled object-oriented imperative programs

As illustrated in Figure 2, a *CLP-decompiled program* consists of a set of *predicates*. A predicate p is defined by one or more clauses which are mutually exclusive. This is ensured, either by means of mutually exclusive *guards*, or by information made explicit on the clause heads (as usual in CLP). Each clause p receives as input a (possibly empty) list of arguments $Args_{in}$ and an input heap H_{in}, and returns the (possibly empty) output $Args_{out}$, a possibly modified output heap H_{out}, and an exception flag. This flag indicates whether the execution ends normally or with an uncaught exception. Clauses adhere to the following grammar. As usual, terminals start by lowercase (or special symbols) and non-terminals by uppercase. Subscripts are provided just for clarity.

$Clause ::= Pred\ (Args_{in},Args_{out},H_{in},H_{out},ExFlag)$:- $[G,]B_1,B_2,\ldots,B_n$.
$G ::= Num^*\ ROp\ Num^* \mid Ref_1^* \backslash== Ref_2^* \mid \mathsf{type}(H,Ref^*,T)$
$B ::= Var\ \#= Num^*\ AOp\ Num^* \mid Pred\ (Args_{in},Args_{out},H_{in},H_{out},ExFlag) \mid$
$\mathsf{new\_object}(H,C^*,Ref^*,H) \mid \mathsf{new\_array}(H,T,Num^*,Ref^*,H) \mid \mathsf{length}(H,Ref^*,Var) \mid$
$\mathsf{get\_field}(H,Ref^*,FSig,Var) \mid \mathsf{set\_field}(H,Ref^*,FSig,Data^*,H) \mid$
$\mathsf{get\_array}(H,Ref^*,Num^*,Var) \mid \mathsf{set\_array}(H,Ref^*,Num^*,Data^*,H)$

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

```
merge([[r(Th),L],[],H_in,H_out,EF) :- get_field(Hin,Th,'SL':first,P1),
        nullcheck1([r(Th),L,P1],[],H_in,H_out,EF).

nullcheck1_1([r(Th),r(L),P1],[],H1,H2,EF) :- get_field(H1,L,'SL':first,P2),
        nullcheck2([r(Th),r(L),P1,P2],[],H1,H2,EF).
nullcheck1_2([r(_),null,_],[],H1,H2,exc(ExRef)) :- new_object(H1,'NPE',ExRef,H2).

nullcheck2_1([r(Th),r(L),r(P1),P2],[],H1,H2,EF) :- get_field(H1,P1,'SL':data,Data1),
        nullcheck3([Data1,r(Th),r(L),r(P1),P2],[],H1,H2,EF).
nullcheck2_2([r(_),r(_),null,_],[],H1,H2,exc(ExRef)) :- new_object(H1,'NPE',ExRef,H2).

nullcheck3_1([D1,r(Th),r(L),r(P1),r(P2)],[],H1,H2,EF) :- get_field(H1,P2,'SL':data,D2),
        if1([D2,D1,r(Th),r(L),r(P1),r(P2)],[],H1,H2,EF).
nullcheck3_2([_,_,r(_),r(_),null],[],H1,H2,exc(ExR)) :- new_object(H1,'NPE',ExR,H2).

if1_1([Data2,Data1,r(Th),r(L),r(P1),r(P2)],[],H1,H3,EF) :- Data1 #> Data2,
        set_field(H1,Th,'SL':first,r(P2),H2), get_field(H2,P2,'SL':next,P2'),
        preloop([r(Th),r(L),r(P1),P2'],[],H2,H3,EF).
if1_1([Data2,Data1,r(Th),r(L),r(P1),r(P2)],[],H1,H2,EF) :- Data1 #=< Data2,
        get_field(H1,P1,'SL':next,P1'), preloop([r(Th),r(L),P1',r(P2)],[],H1,H2,EF).

preloop([r(Th),L,P1,P2],[],H1,H2,EF) :-
        get_field(H1,Th,'SL':first,Curr), loop([r(Th),L,P1,P2,Curr],[],H1,H2,EF).

loop([Th,L,P1,P2,Curr],[],H1,H2,EF) :- loopcond1([Th,L,P1,P2,Curr],[],H1,H2,EF).
loopcond1_1([_,_,null,P2,Curr],[],H1,H2,EF) :- if3([null,P2,Curr],[],H1,H2,EF).
loopcond1_2([Th,L,r(P1),P2,Curr],[],H1,H2,EF) :-
        loopcond2([Th,L,r(P1),P2,Curr],[],H1,H2,EF).

loopcond2_1([_,_,r(P1),null,Cur],[],H1,H2,EF) :- if3([r(P1),null,Cur],[],H1,H2,EF).
loopcond2_2([Th,L,r(P1),r(P2),Curr],[],H1,H2,EF) :-
        loopbody1([Th,L,r(P1),r(P2),Curr],[],H1,H2,EF).

loopbody1([Th,L,r(P1),r(P2),Curr],[],H1,H2,EF) :-
        get_field(H1,P1,'SL':data,Data1), get_field(H1,P2,'SL':data,Data2),
        if2([Data2,Data1,Th,L,r(P1),r(P2),Curr],[],H1,H2,EF).

if2_1([Data2,Data1,Th,L,r(P1),r(P2),r(Curr)],[],H1,H3,EF) :- Data1 #> Data2,
        set_field(H1,Curr,'SL':next,r(P2),H2), get_field(H2,P2,'SL':next,P2'),
        loopbody2([Th,L,r(P1),P2',r(Curr)],[],H2,H3,EF).
if2_2([Data2,Data1,Th,L,r(P1),r(P2),r(Curr)],[],H1,H3,EF) :- Data1 #=< Data2,
        set_field(H1,Curr,'SL':next,r(P1),H2), get_field(H2,P1,'SL':next,P1'),
        loopbody2([Th,L,P1',r(P2),r(Curr)],[],H2,H3,EF).

loopbody2([Th,L,P1,P2,r(Curr)],[],H1,H2,EF) :-
        get_field(H1,Curr,'SL':next,Curr'), loop([Th,L,P1,P2,Curr'],[],H1,H2,EF).

if3_1([r(P1),_,r(Curr)],[],H1,H2,ok) :- set_field(H1,Curr,'SL':next,r(P1),H2).
if3_2([null,P2,r(Curr)],[],H1,H2,ok) :- set_field(H1,Curr,'SL':next,P2,H2).
```

Fig. 2. Working example: CLP-decompiled code.

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

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. Observe that clauses can define both methods which appear in the original source program (*MSig*), or additional predicates which correspond to intermediate blocks in the

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

program (*Block*). An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constraint) variable. Guards might contain: comparisons between numeric data or references and calls to the type predicate, which checks the type of a reference variable (by consulting the heap). Virtual method invocations in the OO language are resolved at compile-time by looking up all possible runtime instances of the method. In the decompiled program, they are translated into a choice of type instructions which check the actual object type, followed by the corresponding method invocation for each runtime instance. Instructions in the body of clauses include: (first row) arithmetic operations, calls to other predicates, (second row) instructions to create objects and arrays, and to consult the array length, (third row) read and write access to object fields, and, (fourth row) read and write access to an array position. As regards exceptions, they can be handled by treating them as additional nodes and arcs in the control flow graph of the program. In our framework, such flows are represented in the CLP-decompiled program with explicit calls to the corresponding exception handlers.

For simplicity, the language does not include features of OO imperative languages like bitwise operations, static fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and booleans. Most of these features can be easily handled in this framework, as shown by the implementation based on actual Java bytecode.

2.2 Semantics of CLP-Decompiled Programs with Heap

When considering a simple imperative language without heap-allocated data structures, like in (Albert *et al.* 2009), CLP-decompiled programs can be executed by using the standard execution mechanism of CLP. In order to extend this approach to a realistic language with dynamic memory, as our first contribution, we provide a suitable representation for the heap and define the heap related operations. Note that, in CLP-decompiled programs the heap is treated as a black-box through its associated operations, therefore it is always a variable. At run-time, the 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. Observe that arrays are stored in the heap together with objects (as it happens e.g. in Java bytecode). Formally, the syntax of the heap at run-time is as follows. The asterisks will be explained later:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{Heap} & ::= & [] \mid [\mathit{Loc}|\mathit{Heap}] & \mathit{Cell} & ::= & \mathit{object}(C^*,\mathit{Fields}^*) \mid \mathit{array}(T^*,\mathit{Num}^*,\mathit{Args}^*) \\ \mathit{Loc} & ::= & (\mathit{Num}^*,\mathit{Cell}) & \mathit{Fields} & ::= & [] \mid [\mathit{field}(\mathit{FN},\mathit{Data}^*)|\mathit{Fields}^*] \end{array}$$

In the upper side of the figure, we present the CLP-implementation of the operations to create heap-allocated data structures (like new_object and new_array) and to read and modify them (like set_field, etc.), and, at the bottom appear some auxiliary predicates. To simplify the presentation some predicates are omitted, namely: build_object/2 resp. build_array/3, which create an object, resp. an array term, new_ref/1 which produces a fresh numeric reference, and subclass/2 which implements the transitive and reflexive *subclass* relation on two classes. member_det/2

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

```
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].

type(H,Ref,T) :- get_cell(H,Ref,Cell), Cell = object(T,_).
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),
                           subclass(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),
                              subclass(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([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([_|RH],Ref,Ob) :- get_cell(RH,Ref,Ob).

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 ground execution.

resp. `replace_det/4` implements the usual deterministic *member*, resp. *replace*, on lists, while `nth0/3` resp. `replace_nth0/3` implements the access to, resp. replacement of, the i*th* element of a list using constraints (multi-moded versions).

We now focus on the *ground* execution of CLP-decompiled programs in which we assume that all input parameters of the predicate to be executed (i.e., $Args_{in}$ and H_{in}) are fully instantiated. The instantiations are provided as constraints in the *input state*. We assume familiarity with the basic notions of CLP. Very briefly, let us recall that the operational semantics of a CLP program P can be defined in terms of *derivations*, which are sequences of reductions between *states* $S_0 \rightarrow_P S_1 \rightarrow_P \ldots \rightarrow_P S_n$, also denoted $S_0 \rightarrow_P^* S_n$, where a *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store θ. If the derivation successfully terminates, then $S_n = \langle \epsilon \mid \theta' \rangle$ and θ' is called the *output state*.

Definition 1 (*ground execution*)
Let M be a method, m be the corresponding predicate from its associated CLP-decompiled program P, and P' be the union of P and the clauses in Figure 3. The *ground execution* of m with input θ is the derivation $S_0 \rightarrow_{P'}^* S_n$, where $S_0 = \langle m(Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag) \mid \theta \rangle$ and θ initializes $Args_{in}$ and H_{in} to be fully ground. If the derivation successfully terminates, then $S_n = \langle \epsilon \mid \theta' \rangle$ and θ' is the output state (ϵ denotes the empty goal).

Every CLP-decompilation must ensure that CLP programs capture the same semantics of the original imperative ones. This is to say that, given a *correct input state*, the CLP-execution yields an *equivalent* output state. By *correct* input state, we mean that all input arguments have the correct types and that the heap has the required contents. For instance, $\theta = \{\mathsf{Args}_{in} = [\mathsf{r}(1),\mathsf{null}] \wedge \mathsf{H}_{in} = [(1,\mathsf{object}(\text{'SL'},[\mathsf{field}(\text{'SL':first},\mathsf{null})]))]\}$ is a correct input state for predicate `merge/5`,

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

whereas $\theta = \{\mathsf{Args}_{in} = [\mathsf{r}(1),\mathsf{r}(2)] \wedge \mathsf{H}_{in} = []\}$ is not correct since the heap does not include the required objects.

Definition 2 (*correct decompilation*)
Consider a method M and a correct input state I. Let m be the CLP-decompiled predicate obtained from M and θ be the input state equivalent to I. If the CLP-decompilation is *correct* then it must hold that, the execution in the OO language of M returns as output state O if and only if the ground execution of m with θ is deterministic and returns an output state θ' equivalent to O.

Correctness must be proven for the particular techniques used to carry out the decompilation. In the interpretive approach, for a simpler bytecode language without heap, Gómez-Zamalloa *et al.* (2009) prove that the execution of the decompiled programs produces the same output state than the execution of the bytecode program in the CLP interpreter. A full proof would require to prove that the CLP interpreter is correct and complete w.r.t the corresponding imperative language semantics. Since our approach is not tied to a particular decompilation technique, in the rest of the paper, for the correctness of our TDG approach, we just require that decompiled programs are correct as stated in Definition 2.

Finally, in the above definition, it can be observed that, since CLP-decompiled programs originate from imperative bytecode, their ground execution is deterministic. The aim of the next section is to be able to execute CLP-decompiled programs symbolically with the input arguments being free variables.

3 Symbolic execution of OO imperative programs

Interestingly, our CLP-decompiled programs can in principle be used, not only to perform ground execution, but also *symbolic execution* (SymEx). Indeed, when the imperative language does not use dynamic memory nor OO features, we can simply run the CLP-decompiled programs by using the standard CLP execution mechanism with all arguments being distinct free variables. For simple imperative languages, this approach was first proposed by Meudec (2001) and developed for a simple bytecode language in Albert *et al.* (2009). However, dealing with dynamic memory and OO features entails further complications, as we show in this section.

3.1 Handling heap-allocation in symbolic execution

In principle, SymEx starts with a fully unknown input state, including a fully unknown heap. Thus, one has to provide some method which builds a heap associated with a given path by using only the constraints induced by the visited code. In the case of TCG, it is required that the ground execution with that heap (and the corresponding input arguments) traverses exactly such path. Existing approaches define novel specific operators to carry out this task. For instance, Charreteur *et al.* (2009) add new constraint models for the heap that extend the basic constraint-based approach without heap. Similarly, Schrijvers *et al.* (2009) provide specific constraints for heap-allocated lists, but needs to adjust the solver to handle other data structures.

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

In our approach, thanks to the explicit representation of the heap, we are able to provide a general solution for the SymEx of programs with arbitrary heap-allocated data structures.

The main point is that in a ground execution, the heap is totally instantiated and, when we execute get_cell/3 (see Figure 3), the reference we are searching for must be a number (not a variable) existing in the heap. In contrast, SymEx deals with partially unknown heaps. Our solution consists in generalizing the definition of get_cell/3 by adding an additional clause (the first one) as follows:

```
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).
```

Intuitively, the heap during SymEx contains two parts: the *known part*, with the cells that have been explicitly created during SymEx 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. Observe the syntax of the heap in Section 2.2 where the *'s indicate where partial information can occur in the heaps during SymEx. Such syntax is hence valid for all heaps appearing at SymEx time. The definition of get_cell/3 now 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 SymEx time can be variables or numbers. (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 use our SymEx framework for the purpose of TCG on our working example. As will be further explained, for this it is required to: (i) impose a termination criterion on SymEx, and (ii) have a mechanism to produce actual values from the obtained path constraints. For (i) let us use **block-k** with $K = 2$. Regarding (ii), we just rely on the *labeling* mechanism of standard *clpfd* domains, since we only get arithmetic path constraints. The rest of the constraints are handled as explained with standard unification through the defined heap operations. Table 1 depicts a graphical representation of the obtained set of test-cases.The table shows, for each test-case, an identifier, a graphical representation of its input and output, and the exception flag. Due to space limitations, we do not show the full input and output heaps, but instead we use the customary graphical representation for the linked lists of integers that they contain (see the example below to understand the correspondence). Let us focus on the first test-case. It corresponds to the following (simplified) sequence of reduction steps $\texttt{merge} \rightarrow \texttt{nullcheck}_1 \rightarrow \texttt{nullcheck}_2 \rightarrow \texttt{nullcheck}_3 \rightarrow \texttt{if1}_1 \rightarrow \texttt{preloop} \rightarrow \texttt{loop} \rightarrow \texttt{loopcond1}_2 \rightarrow \rightarrow \texttt{loopcond2}_1 \rightarrow \texttt{if3}_1$. Its associated answer is $\theta = \{\mathsf{Args}_{in} = [\mathsf{r(Th)},\mathsf{r(L)}] \wedge \mathsf{H}_{in} = [(\mathsf{Th},\ \mathsf{object('SL',[field(first,A)])),\ (L,object('SL',[field(first,B)])),(A,object('SLNode',[field(data,1)])),\ (B,\ object('SLNode',[field(data,0),field(next,null)]))] \wedge \ldots\}$, indicating that merging a list with head "1" and any possible continuation (denoted "C"), and a null-terminated list with head "0", produces an output list with head "0", followed by "1" and followed by the continuation "C".The last three test-cases show that, either if `l` is null, or the

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

Table 1. *Obtained test-cases for working example*

N	Input	Output	EF
1	this.first→(1)→*C* l.first→(0)→null	this.first→(0)→(1)→*C*	ok
2	this.first→(1)→*C* l.first→(0)→(0)→null	this.first→(0)→(0)→(1)→*C*	ok
3	this.first→(1)→null l.first→(0)→(1)→*C*	this.first→(0)→(1)→(1)→*C*	ok
4	this.first→(0)→null l.first→(0)→*C*	this.first→(0)→(0)→*C*	ok
5	this.first→(0)→(1)→*C* l.first→(0)→null	this.first→(0)→(0)→(1)→*C*	ok
6	this.first→(0)→(0)→null l.first→(0)→*C*	this.first→(0)→(0)→(0)→*C*	ok
7	this.first→(0)→*C* l.first = null	-	exc
8	this.first→null l.first→C	-	exc
9	this.first→C l→null	-	exc

`first` field of any of the lists is null, the method throws an exception. This is indeed spotting a bug in the program (assuming it is not the intended behavior).

3.2 Handling pointer aliasing in symbolic execution

A challenge in SymEx of realistic languages is to consider pointer-*aliasing* during the generation of heap-allocated data structures, i.e., the fact that the same memory location can be accessed through several references (called aliases). In the case of TCG, ignoring aliasing can lead to a loss of coverage. Again, our solution consists in further generalizing the definition of get_cell/3 by adding an additional clause (the third one), thus illustrating again the flexibility of our approach:

```
get_cell(H,Ref,Cell)              :-  var(H), !, H = [(Ref,Cell)|_].
get_cell([(Ref',Cell')|_],Ref,Cell)  :-  Ref == Ref', !, Cell = Cell'.
get_cell([(Ref',Cell')|_],Ref,Cell)  :-  var(Ref), var(Ref'), Ref = Ref', Cell = Cell'.
get_cell([_|RH],Ref,Cell)         :-  get_cell(RH,Ref,Cell).
```

Essentially, two cases are distinguished: (a) The reference we are searching for is a number, in that case it must exist in the heap and the 2nd clause will eventually succeed. (b) If Ref is a variable: (b.1) Ref exists in the heap, and the 2nd clause eventually succeeds. Here, Ref must have been already processed (and possible aliases for it might have been created. (b.2) The interesting case is when Ref is a free variable which was not in the heap. In this case, the 2nd clause will never succeed and the 3rd one will unify Ref with all matching references in the heap.

Example 3

Let us consider again the TCG for our working example as in Example 2. Table 2 shows seven additional test-cases obtained using the new definition of get_cell/3. Test-cases 10-12 represent executions in which the two lists to be merged are aliases. The remaining test-cases show other shapes of lists with aliasing among their nodes. In most cases, the result is a cyclic list. This clearly reveals a dangerous behavior of the method which should be controlled by the programmer. Altogether, our set of test-cases provides full coverage w.r.t. the shape of data structures.

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

Table 2. *Additional test-cases when considering pointer-aliasing*

N	Input	Output	EF
10	this.first→0→null l = this	this.first→0	ok
11	this.first→0→0→null l = this	this.first→0→0	ok
12	this.first→null l = this	-	exc
13	this.first→0→null, l.first→0	this.first→0	ok
14	this.first→0→0→null, l.first→0	this.first→0→0	ok
15	this.first→1→null, l.first→0	this.first→0→1	ok
16	this.first→0, l.first→0→null	this.first→0→0	ok

3.3 Inheritance and virtual invocations in symbolic execution

Inheritance and virtual method invocations pose further challenges in SymEx of realistic OO programming languages. From the side of data structure shape coverage, we should create aliasing among objects that possibly have different class types but, due to their inheritance relation, might be aliased at runtime. From the side of path coverage, virtual invocations pose further complications when the object on which the virtual invocation is performed has not been created during SymEx, but is rather accessed from the input arguments. In this case, only the declaration type of the object is known. To achieve path coverage, all implementations of the method that might be invoked at runtime (but not more), should be exercised. Interestingly, our solution solves these issues for free. Let us consider a scenario where we have three classes A, B and C, such that C is a subclass of B, and B a subclass of A; and the following method `m(A a, B b){a.f; b.g; a.p();}`. Let us also assume that both B and C redefine method p. The corresponding CLP-decompiled code contains two calls to get_field/4, resp. with `'A':f` and `'B':g`. During SymEx, the first one will call subclass(X,'A'), which produces three alternatives (X='A', X='B' and X='C'). The second call to get_field will then succeed with cases X='B' and X='C', but fail with X='A'. Thus, the case where a and b are aliased is properly handled, and the calls $B.p()$ and $C.p()$ (and not $A.p()$) will be exercised.

Definition 3 (*symbolic execution*)
Let M be a method, m be the corresponding predicate from its associated CLP-decompiled program P, and P' be the union of P and the clauses in Figure 3 with the described extensions. The *symbolic execution* of m is the derivation tree with root $S_0 = \langle m(Args_{in},Args_{out},H_{in},H_{out},ExFlag) \mid \theta \rangle$ and $\theta = \{\}$ obtained using P'.

The following theorem establishes the correctness of our symbolic execution mechanism. Intuitively, it says that each successful derivation in the symbolic execution produces an output state which is correct, i.e., for any ground instantiation of such derivation we obtain an output state which is an instantiation of the one obtained in the symbolic execution. For simplicity, throughout the paper, we have included in an output state θ two ingredients: the computed answer substitution σ and the actual constraints γ. Given a constraint store θ, we say that σ' is an *instantiation* of θ if

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

$\sigma' \leqslant \sigma$ and $\gamma\sigma'$ is satisfiable. Also, we say that an output state θ' is an instantiation of θ, written $\theta' \leqslant \theta$, when both the corresponding stores and the substitutions hold the $\leqslant$ relation.

Theorem 1 (*correctness*)
Consider a successful derivation of the form: $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow \langle \epsilon \mid \theta \rangle$ which is a branch of the tree with root $S_0 = \langle m(Args_{in},Args_{out},H_{in},H_{out},ExFlag) \mid \{\} \rangle$ obtained in the *symbolic execution* of m. Then, for any instantiation σ' of θ which initializes $Args_{in}$ and H_{in} to be fully ground, it holds that the ground execution of $S'_0 = \langle m(Args_{in},Args_{out},H_{in},H_{out},ExFlag)\sigma' \mid \{\} \rangle$ results in $\langle \epsilon \mid \theta' \rangle$ with $\theta' \leqslant \theta$.

4 (Conditional) TCG of OO imperative programs

An important problem with SymEx, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in general infinite. In the context of TCG, it is therefore essential to establish a *termination criterion*, which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test-cases is generated. In addition to this, some approaches perform *conditional* TCG in which, besides selecting a criterion, the user establishes a precondition which further prunes the evaluation tree. In the remaining of this section, we describe how these issues are handled in our approach.

4.1 Implementing coverage criteria by means of unfolding strategies

A large series of *coverage criteria* (CCs) have been developed over the years which aim at guaranteeing that the program is exercised on interesting control and/or data flows. Applying the coverage criteria on the CLP-decompiled program should achieve the desired coverage on the original bytecode.

Implementing a CC in our approach consists in building a finite (possibly unfinished) evaluation tree by using a non-standard evaluation strategy. In Albert *et al.* (2009), we observed that this is exactly the problem that *unfolding rules* used in partial evaluators of (C)LP solve, and we proposed **block-*k***, a new CC for bytecode which was implemented with the corresponding unfolding rule. In this section, we go further and show that the most common CCs can be integrated in our system using unfolding rules. The following predicate defines a generic unfolding rule for depth-first evaluation strategies which is parametric w.r.t. the CC:

```
unfold(Root,Goal,CCAuxDS,CCParam) :-
(1)   select(Goal,G_left,A,G_right), !,
(2)   (internal(A) -> match(A,Bs) ; (call(A), Bs = []),
(3)   update_ccaux(CCAuxDS,A,CCAuxDS'),
(4)   append([G_left,Bs,G_right],Goal'),
(5)   (terminates(A,CCAuxDS',CCParam) -> add_resultant(Root,Goal')
(6)                                    ; unfold(Root,Goal',CCAuxDS',CCParam)).
unfold(Root,Goal,_,_) :- add_resultant(Root,Goal).
```

The main operation dependent on the CC is `terminates/3`, which indicates when the derivation must be stopped. For this aim, it uses an input set of parameters `CCParam` and an auxiliary data-structure `CCAuxDS`. Intuitively, given a goal `Goal`, an initial `CCAuxDS` and `CCParams`, `unfold/4` performs unfolding steps until either `select/4`

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

fails, because there are no atoms to be reduced in the goal, or `terminates/3` succeeds. In both cases, the corresponding *resultant* is stored, which can then be used to generate a test-case (or a rule in the *test-case generator*; Albert *et al.* 2009). The `Root` argument carries along the root atom of SymEx. An unfolding step consists in the following: (1) select the atom to be reduced, which splits the goal into the selected atom `A` and the sub-goals to its left G_{left} and right G_{right}; (2) match the atom with the head of a clause in the program, or call it in case it is a builtin or constraint; (3) update `CCAuxDS`; (4) compose the new goal; and (5) if the CC stops the derivation (i.e. `terminates/3` succeeds) then store the resultant, otherwise (6) continue unfolding.

In order to instantiate this generic unfolding rule with a specific CC, one has to provide the corresponding auxiliary data-structure and parameters, as well as suitable implementations for `update_ccaux/3` and `terminates/3`. Additionally, `match/2` and `select/4` allows resp. tuning the order of generation of the evaluation tree, and extending the functionality of TCG by allowing *non-leftmost* unfolding steps (Albert *et al.* 2006), as will be further discussed. Note that, in order to guarantee that we get correct results in presence of non-leftmost unfoldings, predicates which are "jumped over" must be *pure* (see Albert *et al.* 2006 for more details). E.g., for **block-***k*, `CCParam` is just the K and `CCAuxDS` is the *ancestor stack* (see Albert *et al.* 2009).

4.2 Including Preconditions during TCG

In practice, it is also essential to prune horizontally the evaluation tree in order to limit the number of test-cases obtained without sacrificing interesting paths. The information used to perform this task is usually provided by the user by means of preconditions on the inputs, formulated using a set of pre-defined properties. These properties can range from simple arithmetic constraints, to more complex properties like *sharing* or *cyclicity* of data-structures. We consider two levels of properties. The first-level comprises properties which can be executed beforehand thus being carried along by the CLP engine, like equality and disequality constraints, arithmetic constraints, etc. E.g., let us re-consider Example 3. We can specify the precondition that the lists are not aliased simply by providing these literals at the beginning of the goal "$\texttt{Args}_{in}$`=[r(Th),L], member(L,[null,r(L')]), Th #\= L'`".

The second level comprises properties that require a certain level of instantiation on inputs in order to be executed. Depending on the property, `unfold/4` can either: perform non-leftmost unfoldings until having the required instantiation, or incrementally check the property as the corresponding structure is being generated, or just delay the property check until the end of the derivation. Interestingly, the different behaviors can be achieved providing suitable implementations of `select/2`. Let us re-consider again Example 3. We can specify the precondition that the lists do not share by providing this in the goal "$\texttt{Args}_{in}$ `= [Th,L], noshare(Th,L)`", where predicate `noshare/2` checks that the data transitively referenced from `Th` do not share with that from `L`.

5 Experimental evaluation

We have implemented and integrated the presented techniques in the PET tool (Albert *et al.* 2010), which is available for download and for online use through

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

Bench	**Es**	**Cs**	**Ms**	**Is**	$\mathbf{T}_{dec}$	$\mathbf{T}_{tcg}^{d50}$	$\mathbf{N}^{d50}$	$\mathbf{C}^{d50}$	$\mathbf{T}_{tcg}^{d200}$	$\mathbf{N}^{d200}$	$\mathbf{C}^{d200}$	$\mathbf{T}_{tcg}^{bk2}$	$\mathbf{N}^{bk2}$	$\mathbf{C}^{bk2}$
Trityp	1	1	1	98	38	22	14	100%	20	14	100%	22	14	100%
Josephus	1	1	3	61	34	6	1	56%	366	45	100%	8	3	100%
DoublyLinkedList	13	2	20	253	157	85	31	37%	594	178	100%	369	116	100%
RedBlackTree	10	2	10	485	365	60	57	30%	2432	539	96%	10010	638	99%
NodeStack	6	3	12	94	51	14	9	100%	8	9	100%	8	9	100%
ArrayStack	7	3	11	103	58	16	15	100%	16	15	100%	16	15	100%
NodeQueue	6	3	15	133	73	18	14	100%	13	15	100%	19	15	100%
NodeDeque	9	3	19	223	150	32	23	67%	38	28	100%	34	28	100%
NodeList	19	9	33	449	383	152	77	73%	182	91	91%	184	91	91%
SortedListPriorityQ	11	14	40	491	442	62	33	29%	190	79	77%	512	164	91%
Sort	4	9	30	735	661	26	12	12%	328	43	44%	400	55	72%

Table 3. *Experimental results*

its web interface at `http://costa.ls.fi.upm.es/pet`. We now present some experiments which aim at illustrating the applicability of our approach to TCG of realistic OO programs. We use two sets of benchmarks. The first group (first four benchmarks) comprises a set of classical programs used to evaluate testing tools taken from (Charreteur and Gotlieb). The second one (last seven) is a selection from the net.datastructures library (Goodrich *et al.* 2003), a well-known library of algorithms and data-structures for Java. Table 3 shows the times taken by the different phases performed by PET as well as the number of test-cases generated and the code coverage achieved for different CCs, **block-***k* and **depth-***k* (which simply limits the number of derivation steps). All times are in milliseconds, and were obtained as the arithmetic mean of five runs on an Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.26 (Debian lenny). For each benchmark we show: the number of methods for which we have generated test-cases (**Es**); the number of reachable classes, methods and Java bytecode instructions (**Cs**, **Ms** and **Is**) (not considering Java libraries); the time taken by PET to decompile the bytecode to CLP ($\mathbf{T}_{dec}$); the time of the TCG, total number of test-cases and *code coverage* for **depth-**50 ($\mathbf{T}_{tcg}^{d50}$, $\mathbf{N}^{d50}$ and $\mathbf{C}^{d50}$); for **depth-**200 ($\mathbf{T}_{tcg}^{d200}$, $\mathbf{N}^{d200}$ and $\mathbf{C}^{d200}$) and for **block-**2 ($\mathbf{T}_{tcg}^{bk2}$, $\mathbf{N}^{bk2}$ and $\mathbf{C}^{bk2}$).

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 usually the main measure considered in TCG to reason about the effectiveness of CCs. We observe that **block-**2 achieves a very high degree of coverage ($\simeq$ 100% for the first 8 benchmarks) thus demonstrating its effectiveness in practice. There are however cases where **block-**2 is not able to achieve 100% coverage. There are different reasons for this: (i) In some cases, $K = 2$ is not sufficient to reach some parts of the code. This is the case of most methods in class Sort. Indeed, **block-**3 achieves 100% of code coverage for this class. (ii) Sometimes there are parts of the code which are simply unreachable at execution time (*dead code*). This is frequent in very generic OO programs, as it is the case of some methods reachable from NodeList and SortedListPriorityQ.

The results obtained for **depth-***k* show that its effectiveness highly depends on the chosen k, and this in turn depends on the particular program. This results in an unsatisfactory CC in practice. E.g., **depth-**50 for Josephus obtains 1 test-case in

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

6 ms, which exercises only the 56% of the code. However **depth-**200 achieves 100% coverage, but at the cost of spending much more time (366 ms), thus obtaining many more test-cases (45). Observe that **block-**2 can achieve 100% coverage with 3 test-cases in only 8 ms.

Overall, from the first group of benchmarks we conclude that PET can compete and even outperform related tools (Charreteur and Gotlieb ; Tillmann and de Halleux 2008). The second group demonstrates the effectiveness of PET with realistic OO programs making extensive use of inheritance and virtual invocations. A careful look at the most complex methods suggests that a more restrictive CC should be used to further prune the SymEx tree when considering more complex programs. E.g. PET obtains 276 (in 880 ms) for RedBlackTree.fixAfterInsertion. We conclude also that the use of preconditions, as explained in Section 4.2, (in principle provided by the user) will be crucial in order to obtain manageable test-suites for more complex programs.

6 Related work and conclusions

In the fields of program verification, static analysis and static checking, transformational approaches are widely used (Vaziri and Jackson 2003; Flanagan 2004). The common technique is to translate an imperative program into an equivalent intermediate representation on which the verification, analysis or checking is performed. The work of Flanagan (2004) is similar to ours in the translation of the imperative program into a constraint logic one. However, the goal here is to perform bounded software model checking rather than TDG and it is not concerned with our problems of ensuring coverage of the shape of data structures. Also, there are no extensions to consider OO features like in our work. In the case of Vaziri and Jackson (2003), the imperative program is translated into a propositional formula and SAT solving is used to find a solution. Again, coverage of shape of data structures is not studied here, which makes it fundamentally different from ours.

Much attention has been devoted to the use of constraint solving in the automation of software testing since the seminal work of DeMillo and Offutt (1991). For the particular case of Java bytecode, Müller *et al.* (2004) develop a symbolic Java virtual machine which integrates constraint solvers and a backtracking mechanism, as without knowledge about the input data, the execution engine might need to execute more than one path. In other approaches the problem is tackled by transforming the program into corresponding constraints, on which the testing process is then carried out by applying constraint solving techniques. Recent progress has been done in this direction towards handling heap-allocated data structures (Gotlieb *et al.* 1998; Charreteur *et al.* 2009; Schrijvers *et al.* 2009). An important advantage of our approach is that, since the source program is transformed into another (constraint logic) program—and not into constraints only—on which the symbolic execution is performed, we can easily track the relation between the test-cases and the source program. Keeping this relation is important for at least two reasons: (1) in order to model new coverage criteria on the source program by using particular evaluation techniques on the CLP counterpart, and (2) to relate the generated

https://doi.org/10.1017/S1471068410000347 Published online by Cambridge University Press

test-cases to paths in the source program to spot errors, etc. This relation is less clear in pure constraint-based approaches (see discussion in Schrijvers *et al.* 2009). Some approaches are focused on improving the efficiency of TDG for dynamic pointer data (Visvanathan and Gupta 2002; Zhao and Li 2007). The basic idea is to separate the process of generating the shape of the data structure to the one of generating values for the fields of data. Our approach is similar to them in that both process are also separated and, although actual experimentation is needed, we believe that a similar efficiency will be achieved.

As another important point, while numeric data can be natively supported by constraint solvers, when extending the constraint-based approach to handle heap-allocated data structures, one has to define new constraint models based on operators that model the heap (Charreteur *et al.* 2009). In Schrijvers *et al.* (2009), these constraints operators are implemented in CHR. In these approaches, one needs to adjust the solver to the particular data structures considered in the language. For instance, Schrijvers *et al.* (2009) provide support for lists and sketches how to extend it to handle trees by adding new operators. Instead, we have provided a general solution to generate arbitrary data structures by means of objects.

Acknowledgements

We gratefully thank Samir Genaim and the anonymous referees for many useful comments and suggestions that greatly helped improve this article.

References

Albert, E., Gómez-Zamalloa, M. and Puebla, G. 2009. Test data generation of bytecode by CLP partial evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*. Lecture Notes in Computer Science, vol. 5438. Springer, 4–23.

Albert, E., Gómez-Zamalloa, M. and Puebla, G. 2010. 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)*. ACM Press, Madrid, 25–28.

Albert, E., Puebla, G. and Gallagher, J. 2006. Non-leftmost unfolding in partial evaluation of logic programs with impure predicates. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Lecture Notes in Computer Science, vol. 3901. Springer, 115–132.

Charreteur, F., Botella, B., and Gotlieb, A. 2009. Modelling dynamic memory management in constraint-based testing. *The Journal of Systems and Software 82,* 11, 1755–1766.

Charreteur, F. and Gotlieb, A. JAUT: A tool for automatic test case generation. `url: http://www.irisa.fr/lande/gotlieb/resources/jaut.html.`

DeMillo, R. A. and Offutt, A. J. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering 17,* 9, 900–910.

Flanagan, C. 2004. Automatic software model checking via constraint logic. *Science of Computer Programming 50,* 1–3, 253–270.

Futamura, Y. 1971. Partial evaluation of computation process—An approach to a compiler-compiler. *Systems, Computers, Controls 2,* 5, 45–50.

Gómez-Zamalloa, M., Albert, E. and Puebla, G. 2009. Decompilation of java bytecode to prolog by partial evaluation. *Information and Software Technology 51*, 1409–1427.

Goodrich, M., Tamassia, R. and Zamore, R. 2003. The net.datastructures package, version 3. Available at `http://net3.datastructures.net`.

Gotlieb, A., Botella, B. and Rueher, M. 1998. Automatic test data generation using constraint solving techniques. *SIGSOFT Software Engineering Notes 23,* 2, 53–62.

Gotlieb, A., Botella, B. and Rueher, M. 2000. A clp framework for computing structural test data. In *Computational Logic*, 399–413.

King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM 19,* 7, 385–394.

Meudec, C. 2001. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability 11,* 2, 81–96.

Müller, R. A., Lembeck, C. and Kuchen, H. 2004. A symbolic java virtual machine for test case generation. In *IASTED Conference on Software Engineering*. IASTED, 365–371.

Schrijvers, T., Degrave, F. and Vanhoof, W. 2009. Towards a framework for constraint-based test case generation. In *19th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'09)*, 128–142.

Tillmann, N. and de Halleux, J. 2008. Pex-white box test generation for .NET. In *Tests and Proofs*. Springer, 134–153.

Vaziri, M. and Jackson, D. 2003. Checking properties of heap-manipulating procedures with a constraint solver. In *TACAS*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. Springer, 505–520.

Visvanathan, S. and Gupta, N. 2002. Generating test data for functions with pointer inputs. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*. IEEE Computer Society, Washington, DC, 149.

Zhao, R. and Li, Q. 2007. Automatic test generation for dynamic data structures. In *SERA '07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*. IEEE Computer Society, Washington, DC, 545–549.

Zhu, H., Hall, P. A. V. and May, J. H. R. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys 29,* 4, 366–427.