A CLP Heap Solver for Test Case Generation

Elvira Albert[1], María García de la Banda[2,5], Miguel Gómez-Zamalloa[1],
José Miguel Rojas[3] and Peter Stuckey[4,5]

[1] *DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain*
[2] *Monash University, Australia*
[3] *DLSIIS, Technical University of Madrid (UPM), E-28660 Boadilla del Monte, Madrid, Spain*
[4] *National ICT Australia, and University of Melbourne, Australia*
[5] *IMDEA Software, Madrid, Spain*

submitted 10 April 2013; revised 24 June 2013; accepted 5 July 2013

One of the main challenges to software testing today is to efficiently handle heap-manipulating programs. These programs often build complex, dynamically allocated data structures during execution and, to ensure reliability, the testing process needs to consider all possible shapes these data structures can take. This creates scalability issues since high (often exponential) numbers of shapes may be built due to the *aliasing* of references. This paper presents a novel *CLP heap solver* for the test case generation of heap-manipulating programs that is more scalable than previous proposals, thanks to the treatment of reference aliasing by means of *disjunction*, and to the use of advanced *back-propagation* of heap related constraints. In addition, the heap solver supports the use of *heap assumptions* to avoid aliasing of data that, though legal, should not be provided as input.

1 Introduction

Software testing (King 1976; Müller et al. 2004; Ammann and Offutt 2008; Tillmann and de Halleux 2008) is the most commonly used technique for validating the quality of software. In practice, software testing is often a mostly manual process that accounts for a large fraction of the software development and maintenance costs. Many techniques have thus been developed to automate software testing by automatically generating test cases that achieve high coverage of the program execution. One of these techniques is *symbolic execution* (King 1976; Gotlieb et al. 2000; Meudec 2001; Müller et al. 2004; Tillmann and de Halleux 2008), which executes programs with symbolic rather than concrete inputs. In doing so, it maintains a path condition that is updated whenever a branch instruction is executed, and encodes the constraints that inputs must satisfy to reach that program point. Test case generation (TCG) is performed by solving the constraints collected for each path using a constraint solver.

One of the main challenges in software testing today is to efficiently handle heap-manipulating programs (Păsăreanu and Visser 2009), that is, programs that create and use dynamically heap-allocated data structures. This is because the testing process needs to consider all possible shapes these data structures can take, in order to ensure reliability. This creates a significant scalability issue for symbolic execution, since an exponential number of shapes may be built due to the *aliasing* of references.

To handle unknown heap structures, existing systems such as PET (Gómez-Zamalloa et al. 2010) and SPF (Păsăreanu and Rungta 2010), use *lazy initialization* (Khurshid et al.

```
1 void m(Ref x, Ref y, Ref z)
2 { x.f=1;
3   z.f=−5;
4   call_a();
5   y.f=x.f+1;
6   call_b();
7   if (x==z) call_c(y.f);
8   else call_d(y.f);
9 }
```

```
10 m_init([r(X),r(Y),r(Z)],[],H_in,H_out) :−
11     m([X,Y,Z],[],H_in,H_out).
12
13 m([X,Y,Z],[],H_in,H_out) :−
14     set_field(H_in,X,f,1,H1),
15     set_field(H1,Z,f,−5,H2),
16     call_a([],[],H2,H3),
17     get_field(H3,X,f,Xf,H4),
18     #=(Yf,Xf+1),
19     set_field(H4,Y,f,Yf,H5),
20     call_b([],[],H5,H6),
21     cond([X,Y,Z],[],H6,H_out).
```

```
22 cond([X,Y,Z],[],H6,H_out) :−
23     ref_eq(H6,X,Z,H7),
24     get_field(H7,Y,f,Yf,H8),
25     call_c([Yf],[],H8,H_out).
26
27 cond([X,Y,Z],[],H6,H_out) :−
28     ref_neq(H6,X,Z,H7),
29     get_field(H7,Y,f,Yf,H8),
30     call_d([Yf],[],H8,H_out).
```

Fig. 1. Motivating example (left). CLP translation (right)

2003). This means that symbolic execution starts with no knowledge of the object values referenced by program variables and, as the program symbolically executes and accesses object fields, it learns the field values in an on-demand fashion. When an unknown integer field is read, a fresh symbol is created for that integer value. When an unknown reference field is read, all possibilities are explored by non-deterministically choosing among the following values: (a) null, (b) any existing symbolic object whose type is compatible with the field's type and might *alias* with it and (c) a fresh symbolic object. Such non-deterministic choices are materialized into branches of the symbolic execution tree. The motivation of this work stems from the observation that branching due to *aliasing* choices can be made "more lazily" than in previous approaches. As we will see, delaying aliasing choices is crucial for the scalability of TCG.

Let us motivate our approach by symbolically executing the m method appearing on the left of Fig.1, assuming that the executions of call_a and call_b do not modify the heap. The symbolic derivation tree computed using standard lazy initialization (as in, e.g., PET and SPF) is shown on the left of Fig.2. Note that before a field is accessed, the execution branches if it can alias with previously accessed fields. For example, the second field access z.f branches in order to consider the possible aliasing with the previously accessed x.f. Similarly, the write access to y.f must consider all possible aliasing choices with the two previous accessed fields x.f and z.f. This ensures that the effect of the field access is known within each branch. For example, in the leftmost branch the statement y.f=x.f+1 assigns -4 to x.f, y.f and z.f, since in that branch all these objects are aliased. The advantage of this approach is that (at least for this program) by the time we reach the if statement we know the result of the test, since each variable is fixed. However, such early branching creates a combinatorial explosion problem since, for example, call_a is symbolically executed in two branches and call_b in five.

Our challenge is to be able to execute symbolically as shown on the rightmost tree of Fig.2, where branching is only produced due to explicit branching in the program, rather than to aliasing. For this purpose, we present a *heap solver* that handles the *disjunction* due to aliasing of references. In particular, at instruction 5 the solver will carry the following conditional information for x.f' (the current value of field f of x): $x = z \rightarrow x.f' = z.f \wedge x \neq z \rightarrow x.f' = x.f$ indicating that if x and z are aliased, then x.f' will take its value from z.f and, otherwise, from x.f. Once the conditional statement at 7 is executed and we learn that x and z are aliased (in the then branch), we need to look up *backwards* in the heap and propagate this unification so that instruction 5 can be fully

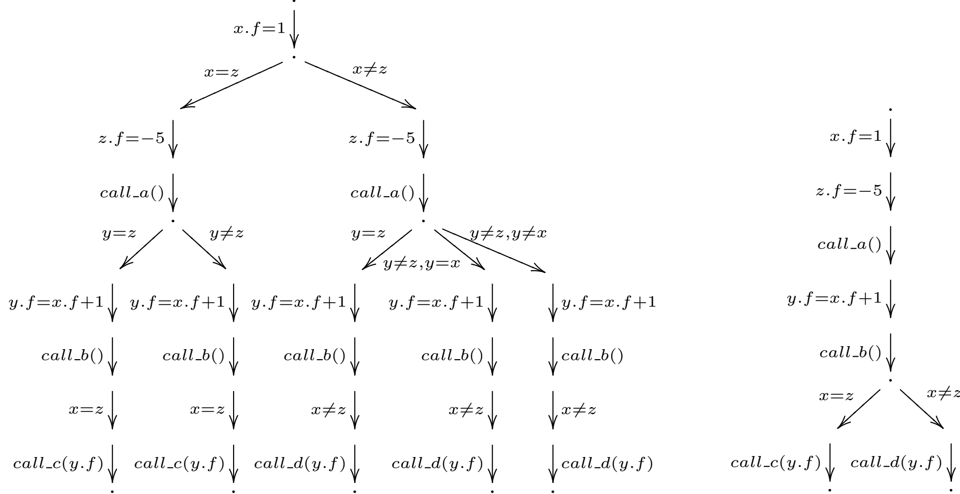

Fig. 2. Symbolic Execution Trees (Left: former approach and Right: our proposal)

executed. This will allow the symbolic execution of call_d(y.f) with a known value for y.f. Our heap solver works on a novel internal representation of the heap that encodes the disjunctive information and easily allows looking up backwards in the heap. In addition, it is possible to provide *heap assumptions* on non-aliasing, non-sharing and acyclicity of heap-allocated data in the initial state. The heap solver takes the assumptions into account to discard aliasing that is known not to occur for some input data. Importantly, our heap solver can be used by any TCG tool for imperative languages through its interface heap operations.

We have integrated our solver in PET and performed an experimental evaluation on methods from the well-known net.datastructures library (Goodrich et al. 2003). Our results demonstrate that our approach can handle heap-manipulating programs efficiently.

Structure of the paper. Section 2 introduces our CLP-based TCG framework. Section 3 develops our heap solver, providing the details of its internal representation, handled operations and propagation mechanisms. Section 4 extends the heap solver with support for heap assumptions. Section 5 reports on implementation and validation results; and finally, Section 6 concludes and summarizes related work.

2 The CLP-based Test Case Generation Framework

CLP-based Test Case Generation (Gómez-Zamalloa et al. 2010) advocates the use of CLP technology to perform test case generation of imperative programs. The process has two main steps. In the first step, the imperative program is automatically transformed into an equivalent *CLP-translated* program in which instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations* (Gómez-Zamalloa et al. 2009). In the second step, the CLP-translated program is executed using the standard CLP execution mechanism, which inherently integrates the use of constraint solvers, and must be extended with a heap solver to handle the heap operations.

A *CLP-translated program* consists of a set of *predicates*, each of them defined by one or more mutually exclusive clauses. Mutual exclusion is ensured either by means of

mutually exclusive *guards*, or by information made explicit on the clause heads (as it is 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 $Args_{out}$ and a possibly modified output heap H_{out}. Clauses adhere to the following grammar where, as usual, terminals start with lowercase (or special symbols), non-terminals with uppercase and subscripts are provided for clarity:

$$
\begin{array}{l}
Clause ::= Pred\ (Args_{in}, Args_{out}, H_{in}, H_{out})\ \texttt{:-}\ [G,] B_1, B_2, \ldots, B_n. \\
\quad Pred ::= Block \mid MSig \qquad Args ::= [\,] \mid [Data|Args] \qquad Data ::= Num \mid Ref \\
\quad\quad B ::= Var \ \#{=}\ Num\ AOp\ Num \mid Pred\ (Args_{in}, Args_{out}, H_{in}, H_{out}) \mid \mathsf{new_object}\ (H_{in}, C, Ref, H_{out}) \\
\quad\quad\quad \mid \mathsf{get_field}\ (H_{in}, Ref, FSig, Var, H_{out}) \mid \mathsf{set_field}\ (H_{in}, Ref, FSig, Data, H_{out}) \\
\quad\quad G ::= Num\ ROp\ Num \mid \mathsf{ref_eq}\ (H_{in}, Ref, Ref', H_{out}) \mid \mathsf{ref_neq}\ (H_{in}, Ref, Ref', H_{out}) \\
\quad Ref ::= null \mid r(Var) \qquad AOp ::= + \mid - \mid * \mid / \mid mod \qquad H ::= Var \\
\quad FSig ::= C{:}FN \qquad ROp ::= \#{>} \mid \#{<} \mid \#{>=} \mid \#{=<} \mid \#{=} \mid \#\backslash{=}
\end{array}
$$

Non-terminals *Block*, *Num*, *Var*, *FN*, *MSig*, *FSig* and *C* denote, resp., the set of predicate names, numbers, variables, field names, method signatures, field signatures and class names. Note that clauses can define methods that appear in the original program (*MSig*), or additional predicates that correspond to intermediate blocks in the program (*Block*). A field signature *FSig* contains the class where the field is defined and the field name *FN*. Guards might contain comparisons between numeric data and between references. In particular, instructions ref_eq and ref_neq check whether *Ref* and *Ref'* are equal and different, respectively. They receive the heap as an explicit parameter since, as we will see later, their execution might modify the heap's contents.

Instructions in the body of clauses include arithmetic operations, calls to other predicates, object creation, and read and write access to object fields. Instruction get_field retrieves from H_{in} the value of the field identified by *FSig* from the object referenced by *Ref*, and returns its value in *Var* leaving the heap unchanged. Instruction set_field sets the field identified by *FSig* from the object referenced by *Ref* to the value *Data*, and returns the modified heap H_{out}. Since subclasses inherit the fields of the class they extend, new_object needs to access the set of classes partially ordered with respect to the *subclass* relation, and the fields declared by each class. Hence, we assume this information is available. Since virtual invocations do not add any complexity to the heap solver, we do not consider them here. Also, for simplicity, the language presented does not include features of OO imperative languages like bitwise operations, static fields, exceptions, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and booleans. However, most of these features can be easily handled by this framework, as shown by our implementation, which handles actual Java bytecode.

Example 1

Fig. 1 shows in the two rightmost columns the CLP-translated code of method m, which was obtained automatically from the bytecode of m. For simplicity, we have omitted the rules that capture the exceptional behaviour when the references are null, since they do not require any special treatment in the heap solver. Thus, our initial predicate, m_{init}, assumes that the three input references are non-null (since they match r(_)) and invokes m. We have also omitted the implicit parameter this of all non-static methods as it does not play any role in the example. The main features that can be observed from the translation are the following. All clauses contain input and output arguments and heaps.

The heap is accessed using the heap operations set_field and get_field. Instruction 5 in the source code is translated into the three CLP instructions 17, 18 and 19. This is because the CLP-translated code is obtained from the bytecode, where addition is performed using three operations: pushing the field value to the stack, increasing it by one and then putting the value again to the heap. Conditional statements in the source program are translated into guarded rules (e.g., cond). Methods (like m) and intermediate blocks (like cond) are uniformly represented by means of predicates and are not distinguishable in the CLP-translated program. Although not present in the example, iteration in the source program is translated into recursion in the CLP program. □

As mentioned before, CLP-translated programs can be executed by using the standard execution mechanism of CLP, given a suitable heap solver. The execution is performed on symbolic values, often represented as *constraint variables*, which are accumulated into path conditions (also called *path constraints*). The path constraints in feasible paths provide pre-conditions on the input data that guarantee the corresponding path will be executed at run-time. Whenever a path constraint is updated, it is checked for satisfiability by the solver. If the path constraint is unsatisfiable, the procedure backtracks. Otherwise, execution continues until a solution (representing a test case) is produced. Collecting all solutions returns the entire set of tests generated for the original program.

In previous work (Gómez-Zamalloa et al. 2010), the heap in the CLP-translated programs was represented explicitly as a list of locations, each being a pair made of a unique reference and a cell. This paper presents a novel approach where the heap is treated as a black-box through its associated operations, which are handled more efficiently by means of a heap solver. As a result, our heap is always represented by a variable. Other constraint-based approaches (Charreteur et al. 2009; Degrave et al. 2010) also represent the heap as a variable, but they differ from us on the definition of the heap operations (see Sec. 6 for a more detailed comparison).

3 The Heap Solver

This section presents our heap solver. In particular, it provides the internal representation of the heap, presents the heap operations, proposes an advanced method for the back-propagation of constraints that allows pruning unfeasible branches earlier, and discusses a simple extension of the heap solver to handle arrays.

3.1 Internal Representation

The heap is internally represented by the heap solver as a tuple $\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle$, where:

- $\mathcal{S}$ is a recursive term that stores every read/write access performed on object fields that have not been explicitly created during symbolic execution, in the order in which such accesses occurred. $\mathcal{S}$ can be seen as a stack of field accesses (the reason for this will become apparent later). Formally, it is defined as $\mathcal{S} ::= getF(Ref, FSig, Var_{\{\mathcal{R}\}}, \mathcal{S}) \mid setF(Ref, FSig, Data, \mathcal{S}) \mid \varnothing$, where Ref, $FSig$ and $Data$ have the same meaning as in the get_field and set_field instructions introduced before, Var is an attributed variable and $\mathcal{R}$ is its attribute, that is, a set of rules representing possible aliasing configurations. Each rule in attribute $\mathcal{R}$ is a conjunction of constraints of the form $\bigwedge_{i=0}^{k-1} Ref \neq Ref_i \wedge Ref = Ref_k \rightarrow Var = Var_k$, corresponding to the aliasing configuration in which if Ref is *only* aliased with Ref_k, then $Var = Var_k$.

- $\mathcal{N}$ is a dictionary that maps fresh numeric references to new objects explicitly created by new_object during the symbolic execution, where an object is a list of fields.
- $\mathcal{RC}$ is a set of disequality constraints over references.

Note that references to objects explicitly created by new_object will appear in $\mathcal{N}$, while references to all other objects will appear in $\mathcal{S}$.

3.2 Heap Operations

The heap solver is accessed by means of its heap operations, get_field, set_field, new_object, ref_eq and ref_neq, which are invoked directly from the CLP-translated program, and by the solve operation which is only invoked at the end of a symbolic execution branch to get one or more concrete solutions. Heap operations, which update the input heap $H_{in} = \langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle$ to obtain $H_{out} = \langle \mathcal{S}', \mathcal{N}', \mathcal{RC}' \rangle$, denoted as $H_{in} \rightsquigarrow H_{out}$ (the conditions for the update will appear over the transition), are handled by the solver as follows:

set_field$(H_{in}, Ref, FSig, Data, H_{out})$**.** If Ref maps to an object O in $\mathcal{N}$, then this operation updates field $FSig$ in O. Otherwise, it adds a new $setF$ element to $\mathcal{S}$, indicating that field $Ref.FSig$ is set to value $Data$.

$$\frac{\begin{array}{c}(Ref \mapsto O) \in \mathcal{N}\\ \mathcal{N}' \leftarrow \mathcal{N}[Ref \mapsto O[FSig \mapsto Data]]\end{array}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}, \mathcal{N}', \mathcal{RC} \rangle} \qquad \frac{\begin{array}{c}(Ref \mapsto O) \notin \mathcal{N}\\ \mathcal{S}' \leftarrow setF(Ref, FSig, Data, \mathcal{S})\end{array}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

get_field$(H_{in}, Ref, FSig, Var_{\{\mathcal{R}\}}, H_{out})$**.** If Ref maps to an object O in $\mathcal{N}$, then this operation simply accesses O, gets the value of field $FSig$ in $Var_{\{\mathcal{R}\}}$, and sets $H_{out} = H_{in}$. Otherwise, a new $getF$ element is added to $\mathcal{S}$ and $Var_{\{\mathcal{R}\}}$ is a fresh variable whose $\mathcal{R}$ attribute and domain are calculated by function ψ.

$$\frac{\begin{array}{c}(Ref \mapsto O) \in \mathcal{N}\\ Var_{\{\mathcal{R}\}} \leftarrow O[FSig]\end{array}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle} \qquad \frac{\begin{array}{c}(Ref \mapsto O) \notin \mathcal{N} \quad \langle \mathcal{R}, dom(Var) \rangle \leftarrow \psi(\mathcal{S}, Ref, FSig, Var, true)\\ \mathcal{S}' \leftarrow getF(Ref, FSig, Var_{\{\mathcal{R}\}}, \mathcal{S})\end{array}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

where $\psi(\mathcal{S}, Ref, FSig, Var, \varphi) =$

$$\begin{cases} \langle \{\varphi \rightarrow Var = F\}, dom(F) \rangle \quad (F\ fresh) & \mathcal{S} = \varnothing & (1)\\ \langle \{\varphi \rightarrow Var = F\}, dom(F) \rangle & \mathcal{S} = [g|s]etF(Ref, FSig, F, \mathcal{S}_i) & (2)\\ \langle \{\varphi \wedge Ref = Ref_i \rightarrow Var = F\} \cup \mathcal{R}_i, dom(F) \cup D_i \rangle & \mathcal{S} = getF(Ref_i, FSig, F, \mathcal{S}_i) & (3)\\ \quad where\ \langle \mathcal{R}_i, D_i \rangle = \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi) & & \\ \langle \{\varphi \wedge Ref = Ref_i \rightarrow Var = F\} \cup \mathcal{R}_i, dom(F) \cup D_i \rangle & \mathcal{S} = setF(Ref_i, FSig, F, \mathcal{S}_i) & (4)\\ \quad where\ \langle \mathcal{R}_i, D_i \rangle = \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi \wedge Ref \neq Ref_i) & & \\ \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi) & \mathcal{S} = [g|s]etF(Ref_i, FSig, F, \mathcal{S}_i) & (5) \end{cases}$$

As an optimization, if $\mathcal{R}$ contains only one single rule $r \equiv true \rightarrow Var = F$, meaning that no aliasing is possible, then $\mathcal{R}$ is emptied and $Var_{\{\mathcal{R}\}} = F$ is added to the store.

ref_eq$(H_{in}, Ref_1, Ref_2, H_{out})$**.** Propagates constraint $Ref_1 = Ref_2$.

$$\frac{t \equiv Ref_1 = Ref_2 \quad \mathcal{S}' \leftarrow propagate(\mathcal{S}, t)}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

propagate returns $\mathcal{S}'$ by simplifying $\mathcal{S}$ w.r.t. a constraint t. Each rule r in attribute $\mathcal{R}$ of each $getF(Ref, FSig, Var_{\{\mathcal{R}\}}, _)$ subterm in $\mathcal{S}$ is treated as follows:

(I) If $lhs(r)$ contains t, we remove t from $lhs(r)$. If the resulting $lhs(r)$ becomes $true$, we add the constraint $rhs(r)$ to the store.

(II) If $lhs(r)$ contains $\neg t$, then $lhs(r)$ can never hold and we remove r from $\mathcal{R}$.

Our CLP implementation of ref_eq simply unifies Ref_1 and Ref_2. This wakes up the constraints in the attributes involving these references so that the above simplification is performed. In addition, we recalculate $dom(Var)$ when rules are removed in (II).

ref_neq*(H_{in},Ref_1,Ref_2,H_{out}).* Propagates constraint $Ref_1 \neq Ref_2$. The disequality is added to the $\mathcal{RC}$ store, since we may later try to add $Ref_1 = Ref_2$ and this must fail.

$$\frac{t \equiv Ref_1 \neq Ref_2 \quad \mathcal{S}' \leftarrow propagate(\mathcal{S}, t) \quad \mathcal{RC}' = \mathcal{RC} \cup \{t\}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \leadsto \langle \mathcal{S}', \mathcal{N}, \mathcal{RC}' \rangle}$$

new_object*(H_{in},C,Ref,H_{out}).* Adds to $\mathcal{N}$ a fresh numeric reference mapped to the newly created object whose fields are initialized to default values (integers are initialized to 0 and references to null).

$$\frac{new(Ref) \quad createObject(C, O) \quad \mathcal{N}' = \mathcal{N} \cup \{Ref \mapsto O\}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \leadsto \langle \mathcal{S}, \mathcal{N}', \mathcal{RC} \rangle}$$

Example 2

Consider the method m whose translation was given on the two right columns of Fig.1. Let us show part of the symbolic execution tree for m([X,Y,Z],[ ],H_{in},H_{out}), starting from the empty heap $H_{in} = \langle \varnothing, \emptyset, \emptyset \rangle$. The following shows, after each $\leadsto_{\mathsf{n}}$ arrow, the update performed to the heap by the execution of the instruction appearing in line n:

$$
\begin{array}{lll}
H_{in} & \leadsto_{14} & H_1 = \langle \mathcal{S}_1, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_1 = setF(X, \mathsf{f}, 1, \varnothing) \\
 & \leadsto_{15} & H_2 = \langle \mathcal{S}_2, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_2 = setF(Z, \mathsf{f}, -5, \mathcal{S}_1) \\
 & \leadsto_{16} & H_3 = \langle \mathcal{S}_3, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_3 = \mathcal{S}_2 \\
 & \leadsto_{17} & H_4 = \langle \mathcal{S}_4, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_4 = getF(X, \mathsf{f}, Xf_{R_4}, \mathcal{S}_3), \\
 & & R_4 \equiv \{X = Z \rightarrow Xf = -5; X \neq Z \rightarrow Xf = 1\} \text{ and } dom(Xf) = \{-5, 1\} \\
 & \leadsto_{18} & dom(Yf) = \{-4, 2\} \\
 & \leadsto_{19} & H_5 = \langle \mathcal{S}_5, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_5 = setF(Y, \mathsf{f}, Yf, \mathcal{S}_4) \\
 & \leadsto_{20} & H_6 = H_5 \\
 & \leadsto_{23} & H_7 = H_6[Xf = -5, Yf = -4] \\
 & \leadsto_{24} & H_8 = \langle \mathcal{S}_8, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_8 = getF(Y, \mathsf{f}, -4, \mathcal{S}_7) \ldots
\end{array}
$$

Note that when executing the get_field in instruction 17, the value stored in the heap for the field is Xf_{R_4}, where Xf is a fresh variable with an associated attribute R_4 built according to the definition of get_field. Namely, we have traversed $\mathcal{S}_3$ down until reaching the $setF$ that has been set in step 14 and we have found two fields $\langle Z, -5 \rangle, \langle X, 1 \rangle$. Thus, we set up the domain of R_4 to $dom(Z) \cup dom(X)$ which results in $dom(Xf) = \{-5, 1\}$, and we add two rules which correspond to cases (4) and (2) in the definition. The first rule is obtained from $\langle Z, -5 \rangle$ and thus has the head $X = Z$, while the second rule is obtained from the negation of the head $X \neq Z$. Observe the role of $getF$ to record a local name for the current value of the field (e.g., Xf for X.f) so that if we look up the same field twice in succession, we get the same name. Instruction 18 simply adds one to the gathered value and stores it in an intermediate variable Yf whose domain is obtained by adding one to $dom(Xf)$. The next interesting step occurs when executing the cond statement. The derivation shows only the branch that applies to the first definition of cond. The execution of the instruction 23 wakes up the definition R_4 and allows us to apply the first rule in R_4, namely as we know that $X = Z$, we can unify Xf to -5. This in turn propagates $Yf = -4$. These unifications are applied to H_6 denoted as $H_6[Xf = -5, Yf = -4]$. Finally, when we apply get_field on Y.f at instruction 24 and traverse the heap, we directly find a $setF$ for Y.f with the value $Yf = -4$. Thus, we do

not traverse it further and call call_c with this fixed argument. Although not shown, the second rule for cond sets $X \neq Z$ and wakes the definition R_4 and sets $Xf = 1$, which in turn propagates $Yf = 2$. The instruction in line 29 retrieves $Yf = 2$ and then calls call_d with this fixed argument.

Assume now that we include the instruction x=new C(); as the first instruction of method m. It will be CLP-translated into new_object(H_{in}, C, X, H_1). According to the definition for new_object, the new object is stored in $\mathcal{N}$. Then, the get_field for x.f will be performed with X being a numeric reference and its value will be retrieved from $\mathcal{N}$. The next instructions will therefore not create (infeasible) aliasings of x with y and z. □

As we can see in the definition of get_field, we only add $Ref \neq Ref_i$ to the lhs if Ref_i arose from a *setF* term. Let us explain this by means of an example. First, consider the fragment "int a=x.f; int b=y.f; int c=z.f". It is sound to have the following attributes associated to Zf in the third get_field instruction, $R \equiv \{Z = Y \rightarrow Zf = Yf; Z = X \rightarrow Zf = Xf\}$, i.e., it is not necessary to include $Z \neq Y$ in the second rule, nor have a rule with $Z \neq Y \wedge Z \neq X \rightarrow Zf = F$. The point is that as the simplified rules are not mutually exclusive, they both apply if $Z = Y = X$, which is correct. The advantage of having unitary guards is that if the body of the rule becomes unsatisfiable, we can negate the head of the rule and propagate this knowledge. This is illustrated in the next section. However, if we add a set_field instruction to the previous fragment "int a=x.f; w.f=a; int b=y.f; int c=z.f", then we must add the non-aliasing guard. We should then have the following attribute for Zf: $R \equiv \{Z = Y \rightarrow Zf = Yf; Z = W \rightarrow Zf = Wf; Z \neq W \wedge Z = X \rightarrow Zf = Xf; Z \neq W \rightarrow Zf = F\}$. This prevents us from applying the third unification of the values of Wf and Xf (which would be incorrect) when Z and W are aliased.

Regarding computational complexity, the heap solver asymptotic complexity is polynomial in the program size. However, this asymptotic complexity is irrelevant since it is dwarfed by the exponential path complexity of symbolic execution. In practice, our experiments show that the practical complexity is acceptable, and more importantly has the effect we want of exponentially reducing the path complexity.

3.3 Backwards Propagation of Constraints

As described in the previous section, our heap solver uses information about equality and disequality of references to determine equality among the heap cells. This is done by propagating such information forwards in the rules of attributes. We can extend the solver straightforwardly to also propagate information backwards. Consider a rule r that defines field F and is part of some attribute $\mathcal{R}$. If $rhs(r)$ is $F = F'$ and the current store implies $F \neq F'$, then r cannot be applied. We can thus recalculate the domain of F by excluding $dom(F')$ from the calculation. Further, if all literals in $lhs(r)$ except one – say l – are known to be true in the current store, we can also assert $\neg l$.

Example 3

Consider the method m but with the condition of the if (in instruction 7) changed to "if (x.f == 1)". This would be translated as get_field(H_6,X,f,$Xf2$,H_7) followed by $Xf2 = 1$. The get_field operation creates propagation rules $X = Y \rightarrow Xf2 = Yf$ and $X \neq Y \rightarrow Xf2 = Xf$. When setting $Xf2 = 1$ the integer solver determines that $Xf2 = Yf$ cannot hold. This means that the solver can propagate $X \neq Y$, which then causes $Xf2 = Xf$. Since we know that $Xf = 1$, the rule $X = Z \rightarrow Xf = -5$ also back propagates to add

the information $X \neq Z$. As a result, we recalculate $dom(Yf) = \{2\}$ and the call call_c is performed with that fixed value for y.f. □

3.4 Extension to Arrays

It is straightforward to extend our language to handle arrays, since we can use the same method as for handling object fields. In this case, the array indices play the role of the references that point to the heap-allocated data.

Example 4
Consider the following fragment of code: " x[k] = -1; x[i] = 2; x[j] = 5; if (x[k] > 0)... " The heap access x[k] is CLP-translated into the operation getArrayElem$(X, k, Xk_{\{\mathcal{R}\}}, H)$ where $Xk_{\{\mathcal{R}\}}$ is an attributed variable built in a similar way to how $F_{\mathcal{R}}$ is built by the get_field instruction. In particular, $\mathcal{R}$ will have the propagation rules $\{K = J \rightarrow Xk = 5; K \neq J \wedge K = I \rightarrow Xk = 2; K \neq J \wedge K \neq I \rightarrow Xk = -1\}$ and $dom(Xk) = \{5, 2, -1\}$. □

Note that this is basically an encoding of the SMT theory of arrays into the same chain of constraints. However, our encoding has an advantage over the SMT approach, since it allows us to later refine the domain of x[k]: if in the above example we later find that $k = i$, we can then refine $dom(Xk)$ to $\{5, 2\}$ (regardless of whether $i = j$ or not).

4 Testing with Heap Assumptions

As we have seen, the TCG process described so far assumes that all possible kinds of aliasing among heap-allocated (reference) input data of the same type can occur. However, it may be the case that while some of these aliasings might indeed occur, others might not (consider, for instance, aliased data structures that cannot be constructed using the public methods in the Java class). In order to avoid generating such inputs, we have extended our framework to handle *heap assumptions*, that is, assertions describing reachability, aliasing, separation and sharing conditions in the heap. We currently support three types of heap assumptions:

- *non-aliasing(a,b):* specifies that memory locations a and b are not the same.
- *non-sharing(a,b):* specifies disjointness, i.e., that references a and b do not share any common region in the heap.
- *acyclic(a):* specifies that a is an acyclic data structure.

Non-aliasing is implicit in the framework and implemented by simply using the constraint $a \neq b$. Non-sharing and acyclicity are properties of the initial heap. In order to implement them correctly we define a CHR predicate initial$(\mathcal{S}, Ref, F)$, which succeeds if field F of Ref refers to the original heap state $\varnothing$. Its implementation, shown in Fig. 3, simply unrolls the $\mathcal{S}$ term ensuring that there is no set_field applied to Ref for field F. Both non-sharing and acyclicity need to track references to the original heap. We assume each call to get_field$(\langle \mathcal{S}, N, RC\rangle, Ref, F, Cell, H')$ creates a call to track_get$(\mathcal{S}, Ref, F, Cell)$ which is implemented as a CHR predicate. Non-sharing of a and b is implemented via the nonsharing(a, b) CHR predicate of Fig. 3, which simply checks that any field reachable in the initial heap from a is not the same as a field reachable from b. Acyclicity of a is implemented via the acyclic(a) CHR predicate, which checks that each path of fields reachable from a cannot reach back to itself.

```
initial(∅,Ref,F)                       <=> true.
initial(getF(_,_,_,S), Ref, F)         <=> initial(S,Ref,F).
initial(setF(Ref,F,_,S), Ref, F)       <=> fail.
initial(setF(Ref1,F1,_,S), Ref, F)     <=> Ref != Ref1 | initial(S,Ref,F).
initial(setF(Ref1,F1,_,S), Ref, F)     <=> F != F1 | initial(S,Ref,F).
track_get(S,Ref,F,Cell)                <=> not initial(S,Ref,F) | true.
---------------------------------------------------------------------------
nonsharing(X,X)                          => fail.
nonsharing(X,Y), track_get(S,X,F,FX)     => initial(S,X,F) | nonsharing(FX,Y).
nonsharing(X,Y), track_get(S,Y,F,FY)     => initial(S,Y,F) | nonsharing(X,FY).
---------------------------------------------------------------------------
acyclic(Ref)                                <=> path(Ref, []).
path(Ref, Path), track_get(S,Ref,F,Cell)     =>  initial(S, Ref, F), member(Ref,Path) | fail.
path(Ref, Path), track_get(S,Ref,F,Cell)     =>  initial(S, Ref, F), notmember(Ref,Path)
                                                 | path(Cell, [Ref|Path]).
```

Fig. 3. Implementation of Heap Assumptions in CHR

Example 5
Consider the pre-condition nonsharing(X, Y) for the method starting with fragment

if (y.f == z && x.f = z && x == z.f) ...

where field f has reference type. The initial get_field on y creates track_get$(\varnothing, Y, \mathsf{f}, Yf)$. This, together with the third rule for nonsharing, adds the constraint nonsharing(X, Yf). The first conjunct forces $Z = Yf$. The set_field of the second conjunct does not create a track_get and does not cause any CHR execution. The third statement creates track_get(S, Z ,f, Zf), which creates nonsharing(X,Zf) using nonsharing(X,Yf). Then $X = Zf$ and the first rule of nonsharing causes failure. Hence, the then branch will not be visited. □

5 Implementation and Experimental Results

We have implemented a prototype of the heap solver and integrated it within PET, an automatic TCG for Java Bytecode available at `http://costa.ls.fi.upm.es/pet`. The prototype is implemented in SWI-Prolog and makes use of its support for attributed variables and of its clpfd library (Wielemaker 2010). All experiments can be reproduced online at the PET web interface by selecting the option "CLP Heap Solver". The implementation provides support for full sequential Java Bytecode. This includes exception handling, which is managed by the transformation from Java Bytecode to CLP and does not require extending the heap solver. Regarding inheritance/polymorphism, realistic Java code makes extensive use of type checks and castings. This is handled during symbolic execution by constraints of the form "subtype of", "not subtype of" and "instance of", over reference types. Our prototype includes a solver to handle such type constraints. In particular, get_field makes use of these type constraints and the solver checks whether the types of two references are compatible before considering their aliasing.

Our experiments aim at illustrating the scalability of our approach on realistic OO programs. They have been performed using as benchmarks a selection of classes from the net.datastructures library (Goodrich et al. 2003), a well-known library of algorithms and data-structures for Java. In particular, we have used as "methods-under-test" the most relevant public methods of the classes *DoublyLinkedList*, *NodeSequence*, *SortedList-PriorityQueue*, *BinarySearchTree* and *HeapPriorityQueue*, abbreviated respectively as *DLL*, *Seq*, *PQ*, *BST* and *HPQ*. Table 1 compares the performance of the new approach (columns labeled with 3) against that of the standard approach based on lazy initial-

Table 1. *Experimental evaluation*

Method	C	N1	T1	U1	N2	T2	U2	N3	T3	U3
DLL.add	139	33	36	1.8k	200	177	7.5k	33	42	2.2k
Seq.removeAt	213	36	42	1.3k	89	116	3.9k	37	50	1.5k
Seq.replaceAt	187	36	41	1.1k	39	47	1.3k	37	27	1.1k
PQ.insert	399	101	160	3.6k	3602	10672	196.7k	101	182	3.6k
PQ.remove	193	12	11	0.4k	86	68	1.2k	12	15	0.4k
BST.addAll	293	379	2019	115.3k	919	-	1832.6k	379	2535	115.3k
BST.find	269	62	108	4.8k	184	285	10.6k	62	98	4.8k
BST.findAll	428	330	2385	120.7k	1165	-	1655.7k	330	2538	120.7k
BST.insert	426	970	2527	84.8k	8365	-	758.5k	970	3924	84.8k
BST.remove	516	203	615	28.7k	2745	5587	200.6k	203	725	28.8k
HPQ.insert	349	61	283	5.4k	135	638	8.3k	95	163	5.2k
HPQ.remove	469	80	1814	69.7k	2021	-	824.8k	146	2378	66.5k

ization, with and without considering reference aliasing (columns labeled with 2 and 1, respectively). For each run we provide the number of clauses in the CLP-translated program (**C**), the number of generated test cases (**N**), the time in milliseconds of the TCG process (**T**) and the number of (thousands of) derivation steps (**U**). For all runs we use the *loop-1* coverage criterion, which limits the number of iterations on loops to at most one. As customary, the test cases are obtained by means of a solve operation which is invoked at the end of each symbolic execution branch and gives a concrete solution. All times are obtained as the arithmetic mean of five runs on an Intel Core i5 CPU at 1.8GHz with 4GB of RAM, running OSX 10.8.2. We use '-' in column **T2** to indicate that the process has not finished within a timeout of 30 seconds. In those cases, **N2** and **U2** correspond to the accumulated numbers when the process is aborted. The figures on columns **N2**, **T2** and **U2** clearly show that the approach based on lazy initialization quickly blows up. By comparing those numbers with **N1**, **T1** and **U1**, where aliasing of references is not being taken into account, we confirm that the explosion on the number of branches is due to the aliasing. Looking at **N3**, **T3** and **U3**, we can observe that our heap solver does not suffer from this explosion problem. Indeed, in terms of speed, in general, our approach (**T3**) can be up to two orders of magnitude faster than the standard approach with aliasing support (**T2**), and similar to (or in some cases even faster than) the standard approach without aliasing support (**T1**). It should be noted that we achieve such speedup even if the back-propagation of constraints is currently only partially implemented. Importantly, observe that the more complex the structure of the program is (in terms of **C** or **N**) the more gain we get with our heap solver.

As regards the number of test cases, the figures in **N3** are in general only slightly greater than those in **N1**. This is because **N3** includes not only the paths of the program that can be reached when there is no aliasing (as **N1**), but also those that can only be reached when some objects are aliased. All additional branches obtained in **N2** are spurious, i.e., they do not lead to further coverage. This becomes apparent in the leftmost tree of Fig. 2 which has five branches (and hence five test cases will be obtained from it), while we only need two branches to have full coverage.

6 Conclusions and Related Work

Ignoring aliasing in TCG leads to loss of coverage and, as a consequence, errors due to undesired aliasing of data go undetected. We have proposed a novel approach to TCG for

heap-manipulating programs which (a) avoids the explosion that occurs when handling aliasing of references and (b) allows specifying initial heap assumptions on acyclicity and disjointness. While we have presented our approach within the CLP-based framework to TCG, where the imperative program in translated to CLP, the heap solver at the core of our approach can indeed be used by other TCG tools by means of its interface heap operations. Therefore, our work is widely applicable.

Constraint-based testing approaches to TCG (Charreteur et al. 2009; Degrave et al. 2010) are closely related to our work. These approaches first extract a constraint system from the source code of the program under test, and then obtain concrete test cases by solving this constraint system. The solvers handle the two sources of non-determinism: the one associated with the control flow (conditional, while statements) and the one associated with the selection of concrete input data (the heap constraints). A main difference with our CLP-based approach is that we represent the control flow by means of a CLP program and handle the constraints associated to heap allocated data by means of the heap solver. This separation of concerns has the advantage that control decisions can be made as soon as possible, while the heap constraints can be lazily executed. Our definition of the symbolic heap operations is different from those of (Charreteur et al. 2009; Degrave et al. 2010). In particular, (Degrave et al. 2010) defines a heap solver using CHRs which is equivalent to the SMT theory of arrays. Since it does not track possible domains of cell lookups as we do, it propagates less information. Note that propagating less information on the heap cells may lead to many infeasible branches which are not pruned until the delayed operations are executed. This could degrade the efficiency. Also their framework is defined only for list-manipulating programs. While they discuss how to extend the approach to new data types, each new data type requires adding new reasoning capabilities, as opposed to our generic approach to memory. The approach of (Charreteur et al. 2009) defines their own constraint operators, but suffers from the severe restriction of being unable to handle inter-procedural calls, which our approach handles transparently. They do not define heap assumption handling. Degrave et al. (2010) mention that their framework could incorporate heap assumptions. We have provided an implementation for the most common heap assumptions, which could be also used in their framework.

While heap assumptions are extensively used in software verification (Podelski et al. 2008), their use in software testing is less common. Notable exceptions are (Visser et al. 2004; Offutt and Liu 1999). In (Visser et al. 2004), user-defined assumptions are given as Java methods that are executed during test case generation. This can be inefficient since the pre-conditions are not given in the same language of the constraint solver (as ours are). Offutt and Liu (1999) use a declarative language to specify preconditions. Our approach is the only one capable of specifying preconditions at the level of the data on which they operate, rather than having heap operations that integrate such preconditions as in (Gómez-Zamalloa et al. 2010). Finally, instead of using constraint operations, we could also use SMT solvers (Peleska et al. 2011). It remains as future work to compare how an SMT-based approach (e.g., (Tillmann and de Halleux 2008)) compares to our pure CLP-based scheme. Note that while the $\mathcal{R}$ attributes mimic the solver for a theory of arrays, our solver keeps track of the disjunction of possible values for base types, without forcing equalities. Hence, it can propagate disequality more strongly than the SMT approach. The backwards propagation explained in Ex. 3 would not occur using an SMT approach (at least until $X \neq Y$ was decided by the solver after further decisions).

Acknowledgments. This work was funded partially by projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900, S2009TIC-1465, ARC DP110102579, and ARC DP110102258.

References

Ammann, P. and Offutt, J. 2008. *Introduction to Software Testing.* Cambridge UP.

Charreteur, F., Botella, B., , and Gotlieb, A. 2009. Modelling Dynamic Memory Management in Constraint-Based Testing. *Journal of Systems and Software 82,* 11, 1755–1766.

Degrave, F., Schrijvers, T., and Vanhoof, W. 2010. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR 2009.* LNCS 6037. Springer, 128–142.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2009. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology 51,* 10, 1409–1427.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2010. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue 10,* 4–6, 659–674.

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. 2000. A CLP Framework for Computing Structural Test Data. In *Computational Logic.* LNAI 1861. Springer, 399–413.

Khurshid, S., Păsăreanu, C. S., and Visser, W. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS.* LNCS 2619. Springer, 553–568.

King, J. C. 1976. Symbolic Execution and Program Testing. *Commun. ACM 19,* 7, 385–394.

Meudec, C. 2001. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab. 11,* 2, 81–96.

Müller, R. A., Lembeck, C., and Kuchen, H. 2004. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTED Conf. on Software Engineering.* ACTA Press, 365–371.

Offutt, A. J. and Liu, S. 1999. Generating Test Data from SOFL Specifications. *Journal of Systems and Software 49,* 1, 49–62.

Peleska, J., Vorobev, E., and Lapschies, F. 2011. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In *NASA FM.* LNCS 6617. Springer, 298–312.

Podelski, A., Rybalchenko, A., and Wies, T. 2008. Heap Assumptions on Demand. In *CAV.* LNCS, vol. 5123. Springer, 314–327.

Păsăreanu, C. S. and Rungta, N. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *ASE.* ACM, 179–180.

Păsăreanu, C. S. and Visser, W. 2009. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf. 11,* 4, 339–353.

Tillmann, N. and de Halleux, J. 2008. Pex: White Box Test Generation for .NET. In *TAP.* LNCS 4966. Springer, 134–153.

Visser, W., Păsăreanu, C. S., and Khurshid, S. 2004. Test Input Generation with Java PathFinder. In *ISSTA.* ACM, 97–107.

Wielemaker, J. 2010. *The SWI-Prolog User's Manual 5.9.9.* Available from `http://www.swi-prolog.org`.