HAL
open science

Constraint-Based Test Input Generation for Java Bytecode

Florence Charreteur, Arnaud Gotlieb

To cite this version:

Florence Charreteur, Arnaud Gotlieb. Constraint-Based Test Input Generation for Java Bytecode. Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10), Nov 2010, San Jose, CA, USA, United States. hal-00699236

HAL Id: hal-00699236

https://inria.hal.science/hal-00699236

Submitted on 21 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Constraint-based test input generation for java bytecode

Florence Charreteur
Universit de Rennes 1
Rennes, France
Florence.Charreteur@irisa.fr

Arnaud Gotlieb
INRIA Rennes Bretagne Atlantique
Rennes, France
Arnaud.Gotlieb@irisa.fr

***Abstract*—In this paper, we introduce a constraint-based reasoning approach to automatically generate test input for Java bytecode programs. Our goal-oriented method aims at building an input state of the Java Virtual Machine (JVM) that can drive program execution towards a given location within the bytecode. An innovative aspect of the method is the definition of a constraint model for each bytecode that allows backward exploration of the bytecode program, and permits to solve complex constraints over the memory shape (e.g., `p == p.next` enforces the creation of a cyclic data structure referenced by `p`). We implemented this constraint-based approach in a prototype tool called JAUT, that can generate input states for programs written in a subset of JVM including integers and references, dynamic-allocated structures, objects inheritance and polymorphism by virtual method call, conditional and backward jumps. Experimental results show that JAUT generate test input for executing locations not reached by other state-of-the-art code-based test input generators such as jCUTE, JTEST and Pex.**

I. Introduction

As integrated within current development environments, automatic test input generation is a promising approach for reducing the cost of software unit testing. In particular, having an approach able to generate and check 100% code coverage of a unit under test is highly desirable for increasing our confidence in the program correctness. However, current realistic solutions for this problem do not achieve complete coverage of usual structural criteria such as all_statements or all_branches. In fact, modern automatic test input generators adequately sacrifice completeness for efficiency. They are able to generate test inputs for programs containing hundreds of thousands lines of code in a couple of minutes but they fail to generate complete statement coverage on simple Java methods that contain cyclic data structures, multi-level dereference aliasing problems, inter-dependent loop statements or non-linear decisions.

Although the underlying reachability problem is undecidable in the general case, it seems that there is room for code-based test input generators applying more costly methods and having the goal of completeness. Constraint-based testing, as introduced by Offutt in 1991 [1], combined symbolic execution and dynamic constraint solving [2] to generate test inputs able to execute specific paths in the code. It was then adapted for C programs and refined with standard constraint programming techniques to target specific locations in the code regardless of the particular path executed [3], [4]. Constraint-based testing was extended to the exhaustive generation of structured inputs for Java programs [5], [6] and the constraint solving of structurally complex constraints [7]. An important work in the area is also the symbolic execution approach of Java PathFinder [8] where "lazy initialization" was proposed [9].

Recent code-based test input generators such as DART [10], PathCrawler [11], CUTE and jCUTE [12], [13] or Pex [14] are based on *dynamic symbolic execution*. They dynamically select a feasible path by picking up a test input and by observing which instructions are executed ; then, they report path conditions by symbolically evaluating the instructions along the path. Finally, by negating the last decision of path conditions and submitting the corresponding system to a constraint or an SMT[1] solver, they try to infer another test input covering a distinct path. When such a test input is found, path coverage of the program is necessarily increased. Whenever path conditions are unsatisfiable, the process backtracks and selects another path to execute. Dynamic symbolic execution performs *forward exploration* as it visits the paths from the entry to the exit and this works fine for quickly exploring a few paths. However, when the goal is to complement an existing test set in order to increase code or branch coverage, this approach is less adapted as forward exploration can be trapped in large subspaces of the path search space. In this paper, we introduce a new constraint-based approach to generate automatically test inputs for Java bytecode programs. Our approach generates a test input under the form of a concrete state of a JVM memory, that can drive the program execution towards a selected bytecode instruction. In the context of unit testing, our approach is made to improve the statements coverage of each method under test thanks to a context-free test case generation. This paper contains two contributions:

1) unlike other approaches [10], [12], [11], [15], [14], our framework explores program paths from the target location towards the program entry point. *Backward exploration* at the bytecode level is not trivial as it requires the development of a kind of inverse reasoning on each bytecode instruction, that is neither available in the SUN's JVM specifications nor in the literature. For that, we developed constraint reasoning on partial abstract memory states. For backward exploration, we also developed a depth-first strategy which takes advantage of early detection of infeasible parts of paths to prune the search space of feasible paths ;
2) a new constraint-based model of the JVM is defined with the notion of *constrained memory variable*. This notion captures abstract memory states and permits to implement deductive rules for a meaningful subset of Java bytecode, including those dealing with objects, inheritance, polymorphism and dynamic memory management. In this model, each bytecode expresses a relation between two *constrained memory variables*, which contain unknown or only partially known variables associated to registers, operands stack or heap. Using constraint propagation and an existing finite domain constraint solver, this model allows decisions that involve

[1] Satisfiability Modulo Theories.

cyclic data structures and reference aliasing problems such as `if( p == p.next )` to be effectively solved.

We implemented our approach in a tool called JAUT (Java Automatic Unit Testing) that can generate input memory states for reaching specific locations within Java bytecode programs. Experimental results on small-sized benchmark programs, including the `TreeMap` Java library of red-black trees, show that JAUT[2] can complement an existing test set with test inputs for locations not covered by other code-based test input generators such as jCUTE[12], [13], JTEST[16] and Pex[14].

The rest of the paper is organized as follows: section 2 introduces our method on a simple example, section 3 presents the memory model we defined to generate test inputs for the JVM, section 4 explains constraint generation and constraint solving for a few bytecodes, section 5 presents the test input generation process which performs backward search across Java bytecode programs, section 6 is dedicated to our experimental validation and discussion of related work, while section 7 concludes and draws perspectives to this work.

II. Detailed example

```
class Coord {
 int x,y;
 Coord(int cx, int cy){ x = cx; y = cy; }

 public Coord moveY(Chrono chrono,int speed) {
  if(chrono.time <= 0 || speed <= 0)
      return this;
  int ytemp = y + chrono.time * speed;
  chrono.time = 0;
  if( ytemp > 65536 )
   {return new Coord(x,65536);}    //instruction i
  else return new Coord(x,ytemp); }

 class Chrono {
 int time;
 ...}
```

Figure 1. Example in Java source code

Consider the Java program of Fig.1 that implements the class `Coord` standing for the Cartesian coordinates in a 2D-space. We selected this simple example just to illustrate the memory model of our constraint-based test input generation approach. Let our test objective be the generation of an input JVM state for method `moveY` that reaches the bytecode corresponding to the instruction i in the source code. Note that we do not pay attention to how the methods of class `Coord` can build such an input state that may be "invalid" if it cannot be reached from other method calls. Object-oriented languages offer ways to directly manipulate the object receiver state without using constructors and accessors (e.g., using *Java reflection* with `setAccessible` or *bytecode translation*) and nothing can guarantee, without additional information, that a given method will not be called from an "invalid input state". Hence testing a program with states that may be invalid is equally important.

The bytecode program shown in Fig.2 corresponds to method `moveY` where bytecode 51 corresponds to the selected test objective.

[2]JAUT and all our experiments are freely accessible at http://www.irisa.fr/lande/gotlieb/resources/jaut.html .

```
public Coord moveY(Chrono, int);
Code: Stack=4, Locals=4, Args_size=3
 0:  aload_1      //push the ref. from the register 1 on the stack
 1:  getfield #4  //update the top of the stack Chrono.time
 4:  ifle 11      //if the top of stack ≤ 0, jump to 11
 7:  iload_2      //push the integer in the register 2 on the stack
 8:  ifgt 13      //if the top of the stack ¿ 0, jump to 13
 11: aload_0
 12: areturn      //return the top of stack
 13: aload_0
 14: getfield #3  //update the top of the stack with the field y
 17: aload_1
 18: getfield #4  //update the top of the stack with Chrono.time
 21: iload_2
 22: imul         //update the top of the stack with the product
                    of the two elements on the top of the stack
 23: iadd
 24: istore_3     //store the top of the stack in the register 3
 25: aload_1
 26: iconst_0     //push the constant 0 on the stack
 27: putfield #4  //update the field Chrono.time
 30: aload_3
 31: ldc #5       //push the integer constant 65536 on the stack
 33: if_icmple 52 //if the second element of the stack is less or
                    equal to the top, jump to 52
 36: ldc #5
 38: istore_3
 39: new #6       //allocate dynamic. mem. for a Coord object
 42: dup          //duplicate the top of stack
 43: iload_3
 44: aload_0
 45: getfield #2  //update the top of the stack with the attr. x
 48: invokespecial #7 //invoke the constructor Coord(int,int)
 51: areturn      //return the top of stack
 ...
```

Figure 2. Example in Bytecodes

Our memory model is based on the notion of constrained memory variables (CMV) which capture JVM states. It contains abstract information on the registers, the operand stack and the heap of the JVM, which can be used to fill in a test script for the method under test.

Our prototype tool JAUT manipulates CMVs and can output, at any moment of the constraint solving process, an excerpt of the CMV associated to `moveY` input JVM state:

```
lin: this   -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[]>,
     speed -< intFD,   [-268435456..268435455]>,
hin:
   0::([Coord],[x-<intFD,[-268435456..268435455]>,
                y-<intFD,[-268435456..268435455]>])
```

Before launching the constraint solving process, the CMV containing the input parameters `lin` and the heap `hin` is automatically generated. `lin` contains two references (`this` and `chrono`) and an integer (`speed`). The object referenced by `this` has already been created on the heap (noted `0::`), as invoking method `moveY` requires a `Coord` object being created. `this` refers to this object as its domain contains a single address (`dom=[0]`). On the contrary, `chrono` does not reference any object for the moment as it can still be null or points to any object of the heap (`dom=all`). `ndom` denotes a set of impossible adresses for a heap reference. Note that references ($0, 1$ and so on) do not reflect any physical counterpart, as our model is abstract and not designed for bytecode execution. Integer variables such as `speed`, `this.x` and `this.y`, that are 32-bits integers in the

Java bytecode program, currently have type `intFD`, with domain[3] $[-268435456..268435455]$.

The constraint generation and solving process of our test input generation method aims at refining this input CMV for method `moveY` by finding values for each variable. We illustrate this process below by showing how this input CMV is successively refined to satisfy the test objective.

Our constraint solving examines each bytecode one by one in a backward fashion. On this program, there is no choice point on the backward exploration as there is only a single path going towards the program entry point. Constraint solving operates with two interleaved processes: *constraint propagation* which uses each constraint to prune the domain of CMV of their inconstistent values and *labeling*, which enumerates the possible values of CMV and launches constraint propagation until no more deduction is possible. Those processes are not new and form the basis of finite domain constraint solving [17]. Let us just recall that all the constraints are added to a constraint propagation queue that iteratively manages each constraint one by one, and each constraint is used to prune the variation domain of its variables. At a given step of the constraint solving process, the following input CMV is produced:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[null,0]>,
     speed -< intFD,   [1..268435455]>,
hin:
  0::([Coord],[x-<intFD,[-268435456..268435455]>,
              y-<intFD,[-268435456..268435455]>])
```

The `ndom` set of reference `chrono` has increased to `[null, 0]`, meaning that `chrono` can neither be null nor points to the object pointed by `this`. These deductions come from the fact that 1) reaching location 51 in the bytecode program requires a non-null value for `chrono` as it is dereferenced several times and 2) `chrono` has to point to a `Chrono` object and not a `Coord` object. The domain of `speed` has been pruned during initial constraint propagation as constraint $speed > 0$ was considered.

Next steps include the creation of object `Chrono` and the start of the *labeling* process for producing a completely instantiated CMV, suitable for test script generation. Suppose that the labeling process has instantiated `speed` to 363 and `this.y` to 5206 using a random labeling heuristic, then we get the following refined input CMV:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     speed -< intFD,   [363]>,
hin:
  0::([Coord],[x-<intFD,[-268435456..268435455]>,
              y-<intFD,[5206]> ]),
  1::([Chrono],[time-<intFD,[167..739477]> ])
```

An object of class `Chrono` has been created and is referenced by `chrono` which has now value 1. Accordingly, the domain of `chrono.time` has been pruned to $[167..739477]$ as the arithmetic constraints $ytemp = this.y + chrono.time * speed, ytemp > 65536$ were considered by the underlying finite domain constraint solver.

However, the input CMV has still uninstantiated variables such as `this.x` and `chrono.time`. Therefore, the labeling process enumerates values for these variables.

So, we get:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     speed -< intFD,   [363]> ,
hin:
 0::([Coord], [x-<intFD,[254]>,y-<intFD,[5206]>]),
 1::([Chrono],[time-<intFD,[167]>])
```

This CMV characterizes an input JVM state that satisfies the test objective of reaching bytecode numbered 51 in method `moveY`. Submitting a test script derived from this state shows that the fault on the converted parameters of `new Coord(ytemp, x)` can be detected.

III. Memory model

Java virtual machine states represent runtime data storage locations such as registers[4], operand stacks and heap data. This section details the representation of data types and data storage locations in our memory model. Note that our framework handles a meaningful subset of Java bytecodes, including integers and references, dynamic allocation and heap-allocated structures, objects inheritance and polymorphism, static and dynamic method invocations, conditional jumps and backward jumps. Arrays accesses and updates are only partially supported and floating-point computations, exceptions, native methods and multithreading are currently left apart.

A. Constraint memory variable

In our memory model, *constrained memory variables* (CMV) are used to represent JVM states. A CMV contains data storage locations where data can be represented by variables along with a domain. Formally, a CMV M is a tuple (F, S, H) where F denotes the set of registers, S denotes the operand stack and H denotes the heap. Note that several distinct CMV M can be created at the same instruction number when loops are present in the bytecode. Each Java bytecode will then be seen as a relation among two CMVs: the CMV M_j before activation of bytecode and the CMV M_k after its activation and before the activation of the following bytecode in the considered sequance of instructions. The tuple (F, S, H) contains variables and domains that are described in the rest of the section.

B. Integer and reference variables

Finite domain variables model integer and reference variables of the program. Their default variation domain depends on the size of their precise type. For instance, the domain $-2^{31}..2^{31}-1$ is associated to an `int` variable. Other integer types are treated accordingly. The default domain of a reference is the *all* symbolic value. This means that the reference can point to every object of the heap. When the solving process prunes the domain of the reference variable, then the domain is composed of a set of integer values, which represent all the heap addresses the reference can point to. Note that the *null* value can also be part of the domain.

[3]There is a current 28-bits limitation for integers in our implementation, due to the use of the SICStus prolog `clpfd` library, but this is not a restriction of the model.

[4]a.k.a. local variables in the SUN JVM specifications.

C. Objects

Each object of the heap is modeled by a pair of elements. The first one, called *type variable*, represents the class of the object, the domain of which is a set of possible classes. This allows to properly handle inheritance and polymorphism. The second element is a mapping associating an integer or a reference variable to each attribute, which corresponds to the value of the attribute. Note that when the domain of the *type variable* contains more than one class, all the possible attributes have to be in the mapping.

For example, if a program defines two classes `A` and `B` where `B` inherits from `A`, and `A` defines attribute `t1` and `B` defines attribute `t2`, then

```
([A,B],[t1-<intFD,[2..5]>,t2-<intFD,[15..17]>])
```

represents an object of class either `A` or `B`, with attribute `t1` necessarily in $2..5$. If the object happens to be of class `B` during the solving process, then its `t2` attribute will belong to $15..17$.

D. Registers and operand stack

In a JVM state, registers are used to store the parameters and the local variables of a method. When the method is dynamic (as opposed to `static` methods), the first register contains the reference to the object (`this`) that calls the method. The operand stack is used to perform the calculations of the method. In a CMV, we use two sequences of variables to represent registers and operand stack. As registers are numbered, the first sequence is a convenient way of implementing an indexed array. On the contrary, the sequence used for representing operand stack is accessed with stack operations. Its first element is considered as its top.

E. Heap

In a CMV, the JVM's heap corresponds to a mapping from a set of addresses to a set of objects, possibly stored at these addresses. We associated a unique integer to each address without taking into consideration the actual physical addresses in the JVM. The domain of a variable H that models the heap is composed of a set of pairs, representing the mapping, and a status. 1) The set of pairs *(address,object)*, denoted by C_H, contains the objects necessarily present in the heap. During the solving process, new pairs can be added to C_H when a dynamic memory allocation bytecode is encountered (i.e., `new`). As our model does not model garbage collection, C_H can only be enriched, reducing the possible states of the heap. 2) The status is either $closed$ or $unclosed$. A $closed$ status of the heap denotes a set of addresses entirely known, even if some objects can still be unknown or only partially known through their domain. On the contrary, an $unclosed$ status denotes states where memory allocation can still enrich the CMV. During the constraint solving process, status can only move from $unclosed$ to $closed$. This notion is introduced to tackle cases where the input memory shape is unknown and we will have to explore the input space during labelling to find it.

IV. Constraint generation and solving

In our framework, constraint generation and solving are performed altogether. This has similarities with the on-the-fly generation of paths of [11] and [14] where path conditions are incrementally tested for satisfiability. We first detail the specification of test objective (Sec. IV-A), and then we present the constraint generation and solving process (Sec. IV-B). We illustrate the constraint generation on an example (Sec. IV-C) and we explain the deduction capabilities of the constraints we have defined to model dynamic memory management (Sec. IV-D).

A. Test objective

Test input generation aims at finding an input CMV that satisfies a given test objective in a Java bytecode method. In our framework, test objectives are specified with the *bytecode instruction number* that corresponds to a bytecode program location. In general, for a given test objective, there are many feasible or infeasible paths that can reach the target locations. Our goal is just to find one input CMV that will drive the computation towards the selected location, whatever is the feasible path executed. Note that such a test objective potentially specifies an infinite set of dynamic locations when the bytecode instruction number is located within loops.

B. Constraint generation from the bytecode

Initially, the input CMV of the method is an unconstrained variable and the constraint generation process will accumulate and solve the constraints, for the current selected path, from the test objective to the program entry point. These constraints capture the pah condition that must be satisfied to follow the current selected path and are used to prune the possible values of the input CMV. Each bytecode can potentially constrain the input CMV variable or intermediate CMV variables, throughout its behavior on the registers, operand stack or heap variables. Based on the semantics of each bytecode as defined in the SUN's specification, we build constraints that implement deductive rules on the CMVs.

Arithmetic bytecodes such as `iadd, ladd, isub, imul, idiv, irem, ineg,...` and comparison bytecodes `if_icmp, if_acmp, lcmp, ...`, act directly on logical variables associated with integers or references, and the elements of the operand stack. They generate arithmetical constraints on the finite domain constraint solver, depending on the type of the operands and the operator. For example, `iadd` generates a relation $Vtemp = Va + Vb$ over two CMVs $M1$ and $M2$, where $Vtemp$ is a fresh finite domain variable associated to the top of the operand stack of $M2$, while Va and Vb are the two finite domain variable associated with the integers on the top of the stack of $M1$. Other arithmetic bytecodes generate similar constraints according to the considered integer type (`int` or `long`) and operator. Comparison bytecodes are handled in a similar way by considering the arithmetic constraint extracted from the condition or from the negation of the condition. This will be made clearer in section V.

Bytecodes for simple constant pool accesses (`ldc, bipush, ...`) or register accesses `iload, aload, iload_<n>, aload_<n>, ...` or register updates `istore, astore, istore_<n>, astore_<n>, ...`, generate equality constraints between the top of the operand stack and registers, depending on the type of the register variable. Equality on reference variables (when `aload` or `astore` is considered) generates a finite domain equality, as well.

To deal with bytecodes for dynamic memory management `new, newarray, ...` and accesses and updates of object fields `getfield, putfield, ...`, special constraints

have to be built. Indeed, these operations maintain complex relations between the CMVs. They can constrain the shapes of heap-allocated data structures and the contents of registers and operand stack. We detail the relations we built for only three of them, as the other can easily be deduced from these ones: $new(class, H0, H1, A)$ maintains the relation between $H0$ the heap of the CMV before execution and $H1$ the CMV after execution, and A a fresh address for the newly created object of type $class$. The relation says that $H1$ is the updated heap $H0$ where reference A points to a newly allocated object.

$getfield(A, Id, H, Val)$ maintains the relation between the heap H, from which an access to an attribute Id of the object designated by reference A is performed and Val a finite domain variable. When reference A has for domain a set of possible references, special deductions can be performed on the possible values for Val. Conversely, using the possible values of Val, special deductions on reference A can be performed. In addition, information on the domains of variables in H allows for deductions on the domains of A and Val. This permits the implementation of powerful forward and backward constraint reasoning.

$putfield(A, Id, Val, H0, H1)$ is more complex as it maintains a relation between $H0$ and $H1$ the heaps of both CMVs, reference A and Val. $H1$ is similar to $H0$ except for object referenced by A where the field Id has been modified to value Val. Possible deductions with this relation are illustrated below.

Bytecodes for arrays manipulation `baload, iaload, iastore, ...` and method invocations `invokespecial, invokevirtual, invokestatic` are taken into account but they are not described here as we considered they were outside the scope of the paper.

C. Example

From the `moveY` method of Fig.2 and the test objective which consists to reach bytecode numbered 51, we get the following constraint system (omitting some details about the calls to constructors). For this example, M_i denotes the memory state before the bytecode instruction number i.

$$
\begin{array}{l}
\{\ F1 = This.Chrono.Speed.Ytemp,\ This \neq null \\
\quad M_{51} = (F1, S1, H1), \\
\quad M_{48} = (F1, X.Y.A1.S1, H2), \\
\qquad\quad invokespecial(Coord, init, [A1, Y, X], H2, H1), \\
\quad M_{45} = (F1, A2.Y.A1.S1, H2), A2 \neq null, getfield(A2, x, H2, X) \\
\quad M_{44} = (F1, Y.A1.S1, H2),\ A2 = this, \\
\quad M_{43} = (F1, A1.S1, H2),\ Y = Ytemp, \\
\quad M_{42} = (F1, A1.S2, H2),\ S1 = A1.S2, \\
\quad M_{39} = (F1, S2, H3),\ A \neq null,\ new(Coord, H3, H2, A1), \\
\quad M_{38} = (F2, Ytemp.S2, H3),\ F2 = This.Chrono.Speed.Ytemp2, \\
\quad M_{36} = (F2, S2, H3),\ Ytemp = 65536, \\
\quad M_{33} = (F2, Val1.Val2.S2, H3),\ Val2 > Val1, \\
\quad M_{31} = (F2, Val2.S2, H3),\ Val1 = 65636, \\
\quad M_{30} = (F2, S2, H3),\ Val2 = Ytemp2, \\
\quad M_{27} = (F2, Val3.A3.S2, H4),\ A3 \neq null, \\
\qquad\quad putfield(A3, time, Val3, H4, H3), \\
\quad M_{26} = (F2, A3.S2, H4),\ Val3 = 0, \\
\quad M_{25} = (F2, S2, H4),\ A3 = Chrono, \\
\quad M_{24} = (F3, Ytemp2.S2, H4),\ F3 = This.Chrono.Speed.Ytemp3, \\
\quad M_{23} = (F3, Val4.Val5.S2, H4),\ Ytemp2 = Val4 + Val5, \\
\quad M_{22} = (F3, Val6.Val7.Val5.S2, H4),\ Val4 = Val6 * Val7, \\
\quad M_{21} = (F3, Val7.Val5.S2, H4),\ Val6 = Speed, \\
\quad M_{18} = (F3, A4.Val5.S2, H4),\ A4 \neq null,\ getfield(A4, time, H4, Val7), \\
\quad M_{17} = (F3, Val5.S2, H4),\ A4 = Chrono, \\
\quad M_{14} = (F3, A5.S2, H4),\ A5 \neq null,\ getfield(A5, y, H4, Val5), \\
\quad M_{13} = (F3, S2, H4),\ A5 = This, \\
\quad M_{8} = (F3, Val8.S2, H4),\ Val8 > 0, \\
\quad M_{7} = (F3, S2, H4),\ Val8 = Speed, \\
\quad M_{4} = (F3, Val9.S2, H4),\ Val9 > 0, \\
\quad M_{1} = (F3, A6.S2, H4),\ A6 \neq null,\ getfield(A6, time, H4, Val9), \\
\quad M_{0} = (F3, S2, H4),\ A6 = Chrono, S2 = \epsilon\}
\end{array}
$$

ϵ is the empty sequence, while $v.s$ denotes the stack s where v is pushed. The elements of the sequence $F2$ represent the parameters and the local variables at the bytecode instruction 51. Instruction 48 calls a constructor and links both CMVs M_{48} and M_{51}. The elements on the top of the stack before instruction are the parameters Y and X as well as the reference $A1$ to the object that calls the method. The heaps $H2$ and $H1$ are linked by the constructor call effect, represented here by a relation $invokespecial$.

The top of the stack before the instruction $getfield$ 45 is a reference $A2$, while the top of stack X after the instruction is the value of the attribute x of the object referenced by $A3$ in the heap $H1$. The relation is maintained by a constraint $getfield(A3, x, H2, X)$. The instruction 39 allocates memory to store a new object of type $Coord$. The relation $new(Coord, H3, H2, A1)$ states that the heap $H2$ contains one added object of type $Coord$. $A1$ is pushed on top of the stack after the instruction. As the only instruction that permits to reach 51 after the conditional instruction 33 is 36, then the top of the stack $A.B$ before the instruction 33 constrains B to be greater than A.

The top of the stack before the instruction 27 contains the value $Val3$ and the reference $A4$. The instruction $putfield$ updates the value of the attribute $time$ with the value $Val3$, for the object referenced by $A3$. $H4$ is the heap before the instruction while $H3$ is the heap after instruction and the constraint $putfield(A3, time, Val3, H4, H3)$ maintains this relation. Finally, the values of the registers $F3$, and the heap $H4$, of the memory M_0, describe a possible test input to reach our test objective 51.

D. Possible deductions with operator putfield

In order to illustrate the behavior of a complex constraint, we show the possible deductions by $putfield$ on a simple example. Consider the heap $H0 = \{(1, (a, [t1 \mapsto V1, t2 \mapsto V2])), (2, (a, [t1 \mapsto V3, t2 \mapsto V4])), (3, (b, [t3 \mapsto V5]))\}$ and the heap $H1 = \{(1, (a, [t1 \mapsto V6, t2 \mapsto V7])),$ $(2, (a, [t1 \mapsto V8, t2 \mapsto V9]), (3, (b, [t3 \mapsto V10])$, which both contain two objects of class a and one object of class b, and the relation $putfield(A, t1, Val, H0, H1)$ where $t1$ is the first of the two attributes of the class a. Class b, which does not inherit from a, has only a single attribute. Suppose that $dom(V1) = [0..10]$, $dom(V3) = [1..3]$, $dom(V8) = 15..2^{31} - 1$ while other domains are unconstrained, let $dom(A) = \{all\}$ and $dom(Val) = [10..40]$, and suppose that status of $H0$ is *unclosed* and status of $H1$ is *closed*. Using the relation $putfield(A, t1, Val, H0, H1)$, several deductions can be performed. 1) As the relation operates on attribute $t1$, one deduces that $dom(A) = all - \{3\}$ as object 3 has only attribute $t3$ and its class b does not inherit from a. Consequently, the following equalities can be added $V2 = V7, V4 = V9$ and $V5 = V10$ as the relation does not modify variables associated with the attributes $t2$ and $t3$. 2)Considering variables $V3$ and $V8$, we see that $dom(V3) \cap dom(V8) = \emptyset$ then one deduces that A refers necessarily to object 2 in the heaps $H0$ and $H1$: $A = 2$. Consequently, $V1 = V6$ is added as the object at the address 1 is not modified and $V8 = Val$ leading to

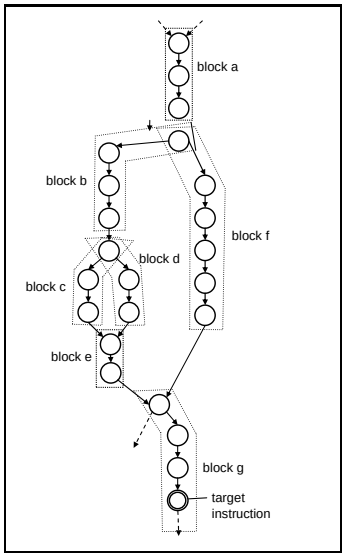

Figure 3. Constraint blocks

$dom(V8) = dom(Val) = [15..40]$. 3) Finally, the *closed* status of $H1$ is propagated to $H0$ as $putfield$ does not add any new object on the heap. So, all the addresses of both heaps are known.

V. Test input generation

A. Backward search

Backward search relies on constraint block accumulation and solving. A *constraint block* in a bytecode program is a set of instructions that is submitted to constraint propagation. The test objective specifies a bytecode to reach, which corresponds to a specific constraint block. So, starting from this block, backward search incrementally accumulates other constraint blocks by trying to find a path towards the program entry point. It is worth noticing that constraint consistency is tested on the fly, each time a new constraint block is added to the constraint system. This permits to quickly detect infeasible parts of program paths. These subpaths should not be confounded with path prefix as they do not begin by the method entry point. When a constraint block has several parent blocks in the control-flow graph, then a choice point is created. Each choice is then explored in a depth-first search until one of them gets to the program entry point. If a choice is shown as being incoherent with the rest of the constraint system, then the process backtracks to the first ancestor choice point and takes an alternative. The strategy gives priorities to shortest intraprocedural subpaths towards the program entry points (i.e., in presence of loops, targeting exit loop constraint blocks). Note however that a parameter has to be set to prevent the process from iterating without terminating in presence of loops.

Let us illustrate why backward search can be interesting w.r.t. forward exploration on the simple example of Fig. 3, where the test objective is to reach constraint block g. Suppose also that block e and block g contain contradictory conditions. For example, e contains assignment `i = 1` while condition to go from block e to block g is `i > 2`. Starting from block g, backward search will first explore subpath $g - e$ by positioning a choice point. As constraint consistency is tested on the fly, the process fails and backtracks to the choice point without exploring the other backward paths starting by $g - e$. Subpath $g - f - a - ...$ is then considered without failing. Constraint inconsistency detection is possible as our framework can reason both in a forward and backward manner, thanks to the use of contraint programming to express relations between CMV. Moreover, our framework implements a precise memory model able to deal with partially known memory states. On this example, forward exploration such as implemented in several dynamic symbolic execution tools would have first considered $a - b - c - e - g$ and failed. Then, backtracking at point b, depth-first forward exploration would have considered $a - b - d - e - g$ and failed again before finding $a - f - g$. To be fair, similar poor behavior can be found for backward exploration as well just by considering incoherent subpaths near the program entry point. We are not saying that backward exploration is better than forward, we just say that both approaches complement each other. Our backward search approach is useful to complement an existing test set by looking at not-covered locations.

B. Test input labeling

When a set of consistent constraint blocks has been found to flow backward from the test objective to the program entry point, every variable of the input CMV has not necessarily been instantiated. A labeling step on the input CMV has then to be launched in order to complement the CMV. The process is recursive and tries to select the best option at each step. It starts by labeling the input formal parameters of the method under test. For references, it first tries $null$, then it tries the addresses of objects in the heap that have compatible type and finally, upon failure, it creates a new object that has a compatible type. Each time a value is assigned to a variable from its domain, the constraints that operate on this variable are awaked and constraint propagation is relaunched. This process can efficiently prune the domain of other variables. When all the input parameter values are fixed, the values of the object attributes have to be labeled in order to build a complete CMV. The process labels first the object attributes that are reachable from the parameters with a single dereferencing level. If the corresponding input state satisfies the test objective, then we get a solution of the problem. On the contrary, upon failure, the process backtracks and the dereferencing level is increased until a given bound, defined by the user. The status is also labeled to the "closed" value in order to indicate that no more object can be added to the input memory state. When the labeling process is finished, then either a complete input CMV is found that reaches the test objective or a failure is reported. Failure may be provoked by several causes, including unreachable test objective, bounds on backward search or dereferencing level, or timeout. Hence, the method is incomplete but note that it can find input cyclic data structures when necessary.

VI. Experimental Validation

A. Implementation

Our prototype tool JAUT (Java Automatic Unit Tesing) takes as inputs a bytecode program given under textual form, obtained with SUN's command `javap` which decompiles the binary bytecode, and a test objective composed of the method name and bytecode instruction number. It reports as output an input CMV, excerpts of which have been presented in Sec. 2 of the paper. JAUT includes 1) a bytecode analysis module that generates a prolog-based internal

structure that can be efficiently explored with backtracks and unification ; 2) a backward search module which is parameterized by: a bound on the number of times certain branches are executed, a bound on the length of paths to be explored, a bound on the dereferencing level and a timeout ; 3) a constraint solver which implements constraint generation and deduction rules for the constraints associated with bytecodes. The constraint solver implements its own constraint propagation queue and its own memory labeling strategy, but it calls the SICStus Prolog `clpfd` library for solving arithmetical constraints.

JAUT handles a meaningful subset of bytecodes, as discussed earlier, but as a research prototype tool, it handles only about a third of the one hundred or so bytecodes of the SUN's JVM specification. Bytecodes that were left apart include bytecodes for integer conversions and low-level shifting, bytecodes for floating-point computations. Note that, for each bytecode, a specific constraint model was developed and deduction rules that captures the operational semantics of the JVM were implemented.

B. Experiments

All the results were computed on a standard single-core machine: an Intel Pentium, 2.16GHZ machine running Windows XP with 2.0GB of RAM. Our experiments aimed at evaluating the capabilities of JAUT to complement an existing test set, obtained by another forward path-exploration test data generator. We selected three such white-box test data generation tools: jCUTE, JTEST version 8.0 and Pex version 0.15.40714.1. Pex is dedicated to .NET but the programs considered in our experiments can easily be compiled to .NET bytecodes[5] Other available test data generators are discussed in the related work section. The goal of the experimental settings was to evaluate the capabilities of JAUT to cover instructions not covered by the three test input generators jCUTE, JTEST and Pex.

Simple programs are used in our experiments: versions of the classical `trityp` and `josephus` programs in bytecodes, methods of the `DoublyLinkedList` and `TreeMap` Java classes. `Trityp` has many infeasible paths while `josephus` [18] manages a cyclic dynamic data structure. We also considered a modified version of this program, called `josephus/m`, where the hard-to-reach decision `ndeEnd.key==41 && nde.key==31` is inserted at the end of the program. The `DoublyLinkedList` class also implements dynamic data structures management and contains input object references and method calls in its code. The `TreeMap` class implements red-black trees which are cyclic structures. The source code of these programs, the source code of JAUT, as well as all the test drivers used for Pex and jCUTE can be found online at `www.irisa.fr/lande/gotlieb/resources/jaut.html`. In all the programs, private fields have been turned into public ones in order, for the three tools, to equally generate valid and invalid input states. For the depth-first backward exploration of JAUT, a bound of 150 bytecodes on the length of path and a bound of 10 for the maximum dereferecing level have been set.

In addition, we considered the following program (in bytecodes) that illustrates a problem related to forward exploration:

```
static int a=1;
public static int foo(int i){
     int j = 10 ;
     while (i > 1){ j++;  i--; }
     if (j > 50*a)
             return 1;  // test objective I
     return 0; }
```

[5]In our results, difference of the .NET runtime cost vs. JVM runtime have been considered neglectible.

Reaching the test objective I implicitely constrains the number of iterations within the loop. On this example, forward exploration will unroll the loop without taking into account the test objective. The parameter `a` can be increased to study the scaling effect of the underlying problem.

C. Results and analysis

On the `foo` program, both JTEST and Pex fail to reach the test objective I. Unlike these tools, jCUTE covers every instructions of the `foo` method, including instruction I, but it takes 10.9sec of CPU time. JAUT also generated an input CMV (corresponding to $i = 42$) that reaches I in 0.15sec of CPU time. Its backward exploration strategy is useful on this example.

We studied the behavior of both jCUTE and JAUT on the `foo` example by increasing the value of `a`. Fig. 4 shows the results of this experiment. The time required by jCUTE increases dramatically on this example as the tool requires an increasing number of trials to satisfy this hard-to-reach test objective. As jCUTE compiles and executes programs to perform dynamic symbolic execution, the CPU time increases accordingly. The JAUT approach is better on this example as constraints are directly handled in the constraint solver and backtracking is hard-coded in Prolog. For other experiments, results are shown in Fig.5 where #Bc is

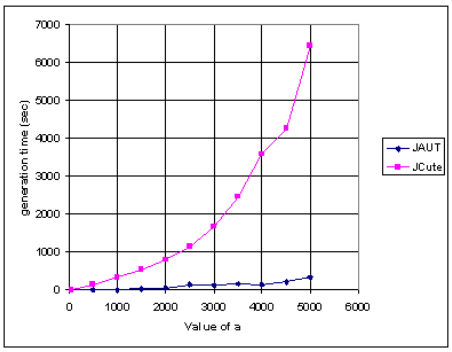

Figure 4. CPU Time to generate input for I in function of a, for method **foo**

the number of bytecode instructions, #To is the number of test objectives. We report the results of jCUTE, Pex, and JAUT to get **complete bytecode instructions coverage** and the results of JAUT to complement the test sets produced by Pex. The fourth tool JTEST reveals itself too poor on this task. In fact, its test input generation strategy is based on the analysis of constants in the program, which prevents many symbolic decisions to be covered. jCUTE and Pex have the primary goal of covering all the feasible paths of the program while JAUT is designed to reach a single instruction in the program. Hence, none of these three tools is optimized to efficiently get complete bytecode instruction coverage. However, in all the cases, there are claims on their ability

to reach instruction coverage. For JAUT, instruction coverage can be automatically reached by launching incrementally requests on not-covered instructions in a script. For the three tools, we report coverage percentage, which we have computed ourselves on a common basis. Indeed, Pex reports dynamic bytecode coverage which does not necessarily correspond to bytecode instruction coverage. So, we computed the coverage by looking at the covered portions of code, rather than taking into account the numbers provided by the tools. We selected two search modes for jCUTE: the former corresponds to depth-first search on the execution tree (both columns noted *jCUTE*) while the second corresponds to random path exploration (both columns noted *jCUTEr*). We measured the CPU time required by JAUT to complement the test set generated by Pex (column *compl. JAUT*) when it was incomplete. When this measure was unnecessary, **n/a** was reported.

For the `trityp` method, experiments show that, unlike jCUTE in both versions, Pex and JAUT succeed to get complete coverage. However, the CPU time required to get this result for JAUT is very long (almost one hour). Indeed, SICStus Prolog needs a long time to demonstrate the insatisfiability of some path condition, as $i + j > k, j + k > i$, $i + j > k$, $j \neq k$, $i \neq k$, $i \neq j$, $k \neq 0$, $j \neq 0$, $i \neq 0$ where i, j, k denote three 32-bit integers, because it requires long-term labeling. Note that another approach would be to use dedicated SMT solver such as Z3 [19] in JAUT to avoid these problems.

Program `josephus` in both versions is hard to cover. The program computes first a dynamic cyclic chain in function of input parameters and then it eliminates one by one the elements by cycling around the chain. Satisfying all the test objectives require unrolling each of the loops 40 times which is not easy to deduce from code analysis only. On this example, Pex obtains very good results by generating 19 test cases while other tools, including JAUT, fail to get complete coverage. Our analysis of the JAUT failure indicates that our depth-first backward search is trapped by the second loop and that it should be refined. This leaves room for improvments.

For the `DoublyLinkedList` class, the three tools succeed to generate complete test sets but JAUT obtains the best CPU time results. Upon analysis of the results for the `add` method, we saw that backward exploration is useful on this example, as forward search requires many paths to be explored. To be fair, it is worth noticing that Pex also tries to cover paths of the called methods while JAUT only considers test objectives within the method under test. In addition jCUTE and Pex also generate test scripts during test input generation, while JAUT do not perform similar generation. On short periods of time, this can have an non-neglectible impact on the results. We did not implement this feature in JAUT as our primary goal was only to demonstrate the interest of fine-grain memory model and backward exploration in constraint-based test input generation.

For the `TreeMap` class, both coverage and CPU time results with JAUT outperform the results of other tools, including Pex. On the `rotateLeft` method, both versions of jCUTE obtained only 20% of bytecode coverage while Pex obtained 88.9%. In fact, covering all the instructions of the method requires explicit solving of the decision `if(p == p.left.parent)` in a certain context, which corresponds to a complex pointer aliasing relation. Note that the time required by JAUT to complement the test set of Pex when covering this decision is very low: 0.05sec. This shows that such decision does not require heavy constraint solving effort. The results obtained for method fixAfterInsertion are the most interesting. Both modes of jCUTE cover only about 20% of the method while Pex covers about 45%. The CPU times required to get these results is rather long for jCUTEr and Pex (resp. 28.30 and 120.00sec). On the contrary, JAUT covers 100% of the method in 22.18sec. When used to complement the Pex test set, it takes 15.49sec, showing that not-covered bytecode instructions are indeed hard to reach. So, JAUT performs well on the methods of the `TreeMap` class as it contains many hard test input generation problems. We interpret this result as a confirmation that using a precise memory model, which in particular deals with reference aliasing, is advantageous for completing an existing test set.

D. Related work

Test input generation at the Bytecode level. JAUT performs constraint-based test input generation from Java bytecode. Therefore, it is mainly related to JPF [8], jCUTE [12], [13], and Pex [14]. Unlike these three tools, JAUT implements backward exploration, meaning that it starts from a target bytecode location and incrementally discovers a feasible path towards the entry. Indeed, JPF, jCUTE and Pex are based on forward symbolic execution which consists to evaluate symbolically the instructions along a path in the same order as execution. Improvements on the search strategy of Pex have been proposed [20] to leverage some of the problems of forward exploration. Note that backward symbolic execution exists for a long time [21], but our approach extends and generalizes this idea to programs containing references on the stack and the heap, as well possibly cyclic dynamic data structures. Reasonning backward at the bytecode level requires a precise memory model to be defined. Our memory model has similarities with those of JPF [8], [9] in that it also implements a form of "lazy initialization". In our approach, decisions such `if( p.next == p )` constrains two variables to be equal and non null without instantiating them. Unlike lazy initialization which makes choices during constraint solving, JAUT reports the choice to the labeling phase. Our memory model also has similarities with the constraint-based models of CUTE [12] and Pex [14] that can deal with symbolic pointer equalities and inequalities, dynamic memory allocation and input data structures. We argued below that JAUT can complement an existing test set produced by one of these tools by applying more costly analysis for specific unreached locations within the bytecode. In fact, the constraint-based reasoning step of JAUT defines precise relations between two CMVs (constraint memory variables) for each bytecode. Of course, the scope of JAUT is currently too restricted (a meaningful subset of JVM bytecodes) to compete with the industrial development of Pex, but we believe that both approaches could complement each other in the long term.

Test input generation for binary programs. OSMOSE [22] and SAGE [23] both implement forward dynamic symbolic execution for binary programs. They handle dynamic jumps and use an untyped memory model able to deal with pointer arithmetic, type casting and multiple dereferencing levels. SAGE builds path conditions that are submitted to the SMT solver Z3 [19]. Unlike SAGE [23], JAUT is based on a constraint propagation and labeling solver. Such solvers are used with success in the context of combinatorial optimization problems as they can solve linear as well as non-linear

Methods	#Bc	#To	jCUTE (cov)	jCUTE (sec)	jCUTEr (cov)	jCUTEr (sec)	Pex (cov)	Pex (sec)	JAUT (cov)	JAUT (sec)	*Compl.JAUT* (sec)
trityp	89	24	83.3%	2.20	87.5%	30.88	100%	4.17	100%	3132.00	n/a
foo	15	3	100%	10.90	75%	18.80	66%	4.51	100%	0.17	0.15
josephus	51	3	100%	1.06	100%	1.06	100%	8.07	100%	0.36	n/a
josephus/m	60	5	70%	192.00	70%	**	100%	8.41	60%	**	n/a
Node class											
insertBefore	36	4	100%	3.02	100%	35.80	100%	4.03	100%	0.20	n/a
DoublyLinkedList											
pop	13	2	100%	2.16	100%	23.00	100%	0.59	100%	0.13	n/a
add	42	4	100%	9.19	88%	45.00	100%	6.09	100%	0.14	n/a
remove	31	4	100%	2.94	100%	1.89	100%	0.69	100%	0.16	n/a
RedBlackTree											
rotateLeft	48	5	20%	26.56	20%	25.10	88.9%	0.65	100%	0.23	0.05
deleteEntry	124	14	50%	2.12	91.6%	5.64	78.6%	56.00	100%	1.44	0.25
fixAfterDeletion	175	11	36.3%	1.11	54.5%	**	81.9%	92.00	100%	7.78	0.88
fixAfterInsertion	127	9	19%	1.30	18.7%	28.30	44.4%	120.00	100%	22.18	15.49

Figure 5. Time and coverage with jCUTE, Pex and JAUT (**: timeout of 3600sec)

problems over the reals or the integers. Variable multiplication or divisions are typical examples of non-linear constraints that are efficiently handled by these solvers. Another advantage of constraint propagation relies on its flexibility to add new user-defined constraints. For JAUT, we built several constraint operators that apply deduction rules to reason over the memory shapes. These operators captures both forward and backward exploration at the same time and propagates domain reductions without instantiating any variable or equality. However, using a SMT-solver such as Z3 [19] to solve arithmetical constraints over fixed-length integer variables would be very interesting to replace `clpfd` in JAUT. Note also that test input generation for binary programs is harder than at the bytecode level as the control flow cannot easily be recovered and variable types are unknown which makes the building of constrained input memory states more complex.

Test input generation from C and Java source code. There are many approaches of automatic test input generation from source code. In our previous works [4], [18], we built memory models for constraint-based test data generator of C programs. Our previous model handled pointers toward named locations of the memory and dynamically allocated structures but they were limited in their scope. Unlike JAUT, the model of [4] did not contain representation of the heap and then it was unable to deal with dynamic memory allocation. The model of [18] was very complex and the generation took too much time. Furthermore, in these preliminary memory models, backward exploration has not been implemented. PathCrawler [11], DART [10], CUTE [12] implement **dynamic symbolic execution** for C source code. On the contrary, JAUT implements **static backward exploration** These two approaches complement each other as dynamic symbolic execution rapidely covers many feasible paths while static backward exploration focusses on hard-to-reach instructions. One advantage of dynamic symbolic execution is that it can deal with third-party libraries or calls to native methods while static backward exploration cannot as there is no available constraint model. Another difference concerns the capabilities of dynamic symbolic exploration to use concrete value instead of symbolic ones for simplifying constraint solving. Note that the constraint reasonning model of JAUT handles complex constraints and just reports value choices to the labeling phase. EXE [15] implements global symbolic evaluation by launching an eager path exploration of the C program under test. Although this approach is appealing if implemented on a grid platform, it can reveal disastrous for programs that contain a huge number of paths. As soon as a program contains a loop, its number of paths is unbounded and even when it contains only conditionals, its number of path grows exponentially with the number of decisions in the worst case. Unlike JAUT, EXE does not use incremental constraint solving, so inconsistencies due to infeasible paths may be lately discovered and this could penalize the test input generation if implemented on a single machine. Note that EXE uses the STP SMT-solver which won several competitions and that it implements nice optimizations such as constraint caching and symbolic memory accesses tracking. However, as pointed out by the authors, its memory model does not handle double pointer dereferencing levels. The memory model of JAUT handles multiple dereferencing levels, up to a user-defined bound. Exhaustive bounded testing of Java programs as implemented in TestEra [5] and KORAT [6] is a test input generation method that exhaustively explores the input search space. The approach can generate a large number of dynamically allocated structures but, unlike JAUT, it does not target specific locations or paths in the code. In fact, these constraint-based approaches have developed complex strategies to efficiently generate test inputs [7] but they cannot solve path constraints extracted from programs.

Counter-example generation. Software model-checkers such as Save [24], Blast [25], Magic [26] or Cbmc [27] explore the paths of a bounded model of C programs in order to find a counter-example path to a temporal property. Some of them also address *statement reacheability* by generating test inputs to reach specific locations within the source code [8]. Some of them exploit *predicate abstraction* to boost the exploration in the context of CEGAR that stands for (Counter-Example Generation through Abstraction Refinement). JAUT contrasts with these model-checkers and CEGAR as it does not abstract the program and does not generate spurious counter-example paths. In particular JAUT builds a constraint model of bytecode program by capturing an error-free concrete semantics without considering a boolean abstraction of the program structure. On the one hand, this allows for precise input memory state to be built but, on the other hand, requires costly analysis to be implemented.

VII. Conclusions and further work

In this paper, we proposed a new constraint-based test input generation approach for testing Java programs at the bytecode

level. We developed a logical memory model for a meaningful subset of bytecodes and proposed deductive rules to solve complex constraints such as `p == p.next`. Unlike other approaches, our prototype tool JAUT implements depth-first backward search and our experimental results indicate that this strategy complements forward search test data generation. Several short term improvements include the use of more dedicated constraint solvers to solve arithmetical constraints of the path conditions (e.g., Z3). Backward search could also be refined with an iterative bounded depth-first strategy to improve our handling of nested and sequential loop computations. Finally, our memory model could be improved by considering more fine-tuned awakening conditions. In the long term, our memory model could be completed to deal with exceptions as they just represent new control flow structures. Multi-threading also appears as an essential topic in Java programming and implementing a backward search strategy for test input generation of multi-threaded bytecode programs would certainly be beneficial to the community.

REFERENCES

[1] R. DeMillo and J. Offut, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[2] J. Offut, Z. Jin, and P. J., "The dynamic domain reduction procedure for test data generation," *Software–Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[3] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'98)*, 1998, pp. 53–62.

[4] A. Gotlieb, T. Denmat, and B. Botella, "Goal-oriented test data generation for pointer programs," *Information and Soft. Technol.*, vol. 49, no. 9-10, pp. 1030–1044, Sep. 2007.

[5] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proc. of the 16th IEEE int. conf. on Automated soft. eng. (ASE'01)*, 2001, p. 22.

[6] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proc. of the int. symp. on Soft. testing and analysis (ISSTA'02)*, 2002, pp. 123–133.

[7] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient solving of structural constraints," in *Proc. of the int. symp. on Software testing and analysis (ISSTA'08)*, 2008, pp. 39–50.

[8] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation in java pathfinder," in *Proc. of ISSTA'04*, 2004.

[9] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transfer*, vol. 11, pp. 339–353, 2009.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proc. of PLDI'05*, 2005, pp. 213–223.

[11] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proc. Dependable Computing - EDCC'05*, 2005.

[12] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proc. of ESEC/FSE-13*. ACM Press, 2005, pp. 263–272.

[13] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *18th Int. Conf. on Computer Aided Verification, (CAV'06)*, ser. LNCS 4144, 2006, pp. 419–423.

[14] N. Tillmann and J. de Halleux, "Pex: White box test generation for .net," in *Proc. of the 2nd Int. Conf. on Tests and Proofs*, ser. LNCS 4966, 2008, pp. 134–153.

[15] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "Exe: automatically generating inputs of death," in *Proc. of Comp. and Communications Security (CCS'06)*, 2006, pp. 322–335.

[16] R. Grehan, "Jtest continues its trek toward code-testing supremacy," in *InfoWorld*, Oct. 2006.

[17] K. Marriott and P. Stuckey, *Programming with Constraints : An Introduction*. The MIT Press, 1998.

[18] F. Charreteur, B. Botella, and A. Gotlieb, "Modelling dynamic memory management in constraint-based testing," in *TAIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.

[19] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of TACAS'08, an ETAPS conference*, Apr. 2008, pp. 337–340.

[20] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. the 39th Int. Conf. on Dependable Systems and Networks (DSN'09)*, Jun. 2009.

[21] S. Muchnick and N. Jones, *Program Flow Analysis: Theory and Applications – Chapter 9 : L. Clarke, D. Richardson*. Prentice-Hall, 1981.

[22] S. Bardin and P. Herrmann, "Structural testing of executables," in *1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08)*, 2008, pp. 22–31.

[23] B. Elkarablieh, P. Godefroid, and M. Levin, "Precise pointer reasoning for dynamic test generation," in *Proc. of ISSTA*, 2009.

[24] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze, "Using symbolic execution for verifying safety-critical systems," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*. Vienna, Austria: ACM, September 2001, pp. 142–150.

[25] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. of 10th Workshop on Model Checking of Software (SPIN)*, 2003, pp. 235–239.

[26] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. on Soft. Eng. (TSE)*, vol. 30, no. 6, pp. 388–402, June 2004.

[27] E. Clarke and D. Kroening, "Hardware verification using ANSI-C programs as a reference," in *Proc. of ASP-DAC'03*, Jan. 2003, pp. 308–311.