Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 48 (2006) 1187–1198
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Code-coverage guided prioritized test generation

J. Jenny Li [a,*], David Weiss [a], Howell Yee [b]

[a] Avaya Laboratories Research, Software Techonology Research, 233 Mt. Airy Road, Basking Ridge, NJ 07920, USA
[b] Lincoln Laboratory, Massachusetts Institute of Technology, 244 Wood Street Lexington, MA 02420-9108, USA

Received 12 April 2006; accepted 14 June 2006
Available online 9 August 2006

Abstract

Most automatic test generation research focuses on generation of test data from pre-selected program paths or input domains or program specifications. This paper presents a methodology for a full solution to code-coverage-based test case generation, which includes code coverage-based path selection, test data generation and actual test case representation in program's original languages. We implemented this method in an automatic testing framework, eXVantage. Experimental results and industrial trials show that the framework is able to generate tests to achieve program line coverage from 20% to 98% with reduced overall testing effort. Our major contributions include an innovative coverage-based program prioritization algorithm, a novel path selection algorithm that takes into consideration program priority and functional calling relationship, and a constraint solver for test data generation that derives constraints from bytecode and solves complex constraints involving strings and dynamic objects.

Keywords: Test generation; Constraint solver; Code-coverage

1. Introduction

Unit testing has become an important step in software development. It is used in both extreme programming and conventional programming. It promises to move the costly testing and defect removal activities to earlier stages of software development, thus reducing their costs. Writing unit tests is not just a good habit, but also an essential part of the internal deliverables. Omitting the unit tests is like designing a car engine and installing it into the finished prototype untested, which nobody in the car industry would do. On the other hand, unit test code is often not part of the deliverable code that gets delivered to the customer. Sometimes it is difficult to justify spending as much time in writing tests as writing code for a customer. Therefore, it is important to reduce the effort of unit testing by using automation, so that unit testing can be more widely adapted by developers.

Many parts of unit testing have been automated. For example, since the unit tests are often represented in the source code's language, they can be compiled with the source and executed automatically. Generation of unit testing frameworks is also automated (e.g., Junit and Cunit). However, the generated tests are represented in mocks or stubs, where users still need to fill in detailed algorithms. Furthermore, none of the existing generation methods emphasize generating efficient test cases to increase the code-coverage in an effective way. On the other hand, coverage-based testing tools do not consider automatic test generation. Even though some, such as χSuds [1] provide hints on which part of the code should be tested first, they fail to go one step further, i.e., they fail to generate the test sequence, let alone the actual test cases. The work in this paper attempts to combine the two research area, code-coverage testing and automatic test generation, through an automatic testing framework.

The paper presents an automation framework that computes the priorities of the program under test based on code-coverage potential and automatically generates tests to cover high-priority points. Fig. 1 shows workflows of

* Corresponding author. Tel.: +1 908 696 5147; fax: +1 908 696 5402.
E-mail addresses: jjli@research.avayalabs.com (J.J. Li), weiss@research.avayalabs.com (D. Weiss), hyee@ll.mit.edu (H. Yee).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.06.007

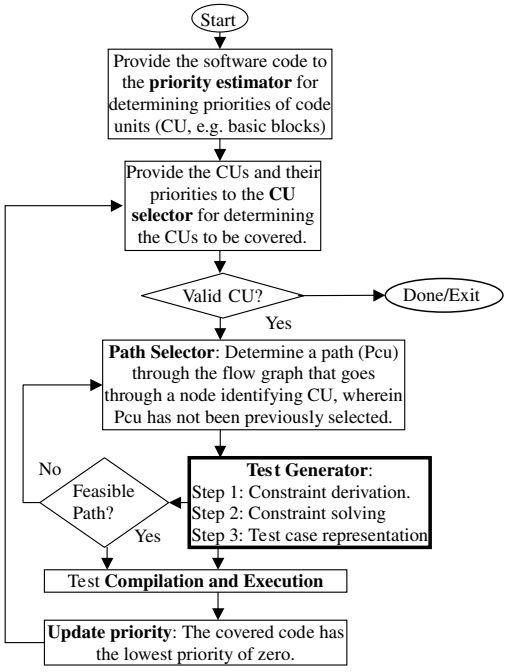


Fig. 1. A framework of automatic unit testing.

the framework, including artifacts such as estimating code-coverage priority, selecting high-priority code unit (**CU**), selecting a path going through the high-priority **CU**, automatically generating tests using a constraint solver, compiling and executing tests, and updating testing result and coverage reports. The focus of this paper is the coverage-based test generator that uses constraint solvers to find suitable test data of which the execution will traverse the selected code sequence.

The test generation box includes three steps: (1) constraint derivation based on the selected sequence, (2) constraint solving to find parameter and class attributes' initial values, and (3) constructing test cases in the original program language. Existing methods that use constraint solvers to generate test data often assume that the paths are pre-selected. We have developed an effective method that selects prioritized paths, including loops, to efficiently increase code coverage, as reported in [2]. Existing constraint solvers deal only with numeric data types such as integer, float, Boolean, and their arrays [3]. Our solver moves one step ahead to handle variables of string types, regular expressions, and objects. This extension allows our solver to generate tests for real-life industrial software applications.

Much research on automatic test generation is based on specifications/models other than source code. For example, studies in [4] have applied control flow and data flow-based test selection criteria to system specifications in SDL for generating tests. Similar research has also been conducted on how to generate tests from UML models [5–7], FSM/EFSM/CEFSM-based models [8,9], and combinatorial designs [10]. While a model-based method may be suitable for system level testing, it is not practical for unit testing because of the high cost in writing an additional model for each source unit. For example, if the unit is a class method that sets the value of one class attribute. It seems excessive to write a model to test such a simple method, because the model itself maybe more complex than the actual code.

Unlike specification-based test generation, code-based test generation does not generate testing oracles for validating test outputs. Much research has been conducted on testing oracles and their generations, which are not the focus of this paper. The other focus of this paper is Java bytecode based methods directly applicable to many web applications.

The rest of the paper is organized as follows. Section 2 briefly describes the steps leading to the automatic test generator. Section 3 presents our constraint solving method based on Java bytecode for automatic test generation, including detail description of the method and its implementation in an automatic testing framework and some experimental results. An overview of related studies appears in Section 4. Our conclusion and future research directions are in Section 5.

2. Steps proceeding test generator

The focus of this paper is on the highlighted box of automatic test generation in Fig. 1. This section briefly discusses the steps leading to this box. First, the code units are prioritized based on various criteria, whose algorithms are given in [2]. With the calculations of the code prioritization algorithm, each node (code unit) of the program has been assigned a priority, and a set of hot-spots (highest priority points) can be identified. The tests covering the hot-spot will guarantee to cover the most number of total nodes, including the function the hot-spot resides and functions called by this function. Now the question is how to identify sequences of nodes, starting from the entry point, going through the hotspots, and ending at an exit point. Because of loops, such sequences should be represented in sequence patterns, i.e., production rules similar to regular expressions. We use symbols "<" and ">" to denote sequences. Path selection includes four tasks: (1) identifying loops, including their entry points and returning points, and denoting loops with symbols like L_1, L_2,..., and L_n, where n is the total number of loops; (2) identifying simple sequences (going through loops once) from the starting point going through the hot-spots, and ending at an exit point; (3) deriving sequence patterns by substituting loops with multiple alternative passing of the loops; and (4) generating sequences in a priority order. We use an example control flow graph, as given in Fig. 2, to illustrate these steps.

(1) *Identifying loops*. We use a modified "topological sort" [11] algorithm to detect and mark the entry and returning points of loops. Starting from the beginning node of a program, check whether the current node has incoming edges. If so, remove and record them in a potential loop

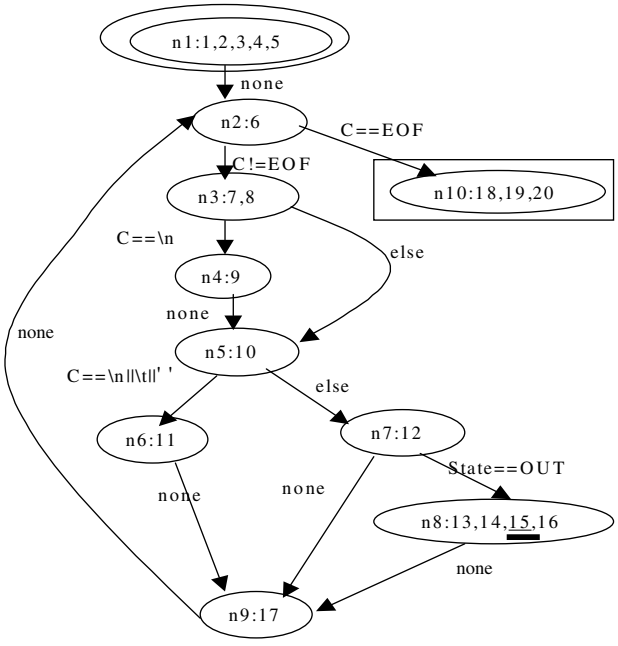


Fig. 2. A control flow graph example.

list, which includes information of loop entry, i.e., current node; and loop returning node, i.e., the node connected to the other end of an incoming edge. Then remove the current node and its corresponding outgoing edges and select one of the nodes without any incoming edge to be the current node. If all nodes have at least one incoming edge, select one of the nodes connected to the just removed node. Repeat the checking again for the new current node. For the graph in Fig. 2, sorting starts from n_1. It first removes n_1 and moves on to n_2. Since n_2 has one incoming edge, remove the edge and mark n_2 as the entry point and n_9 as the returning point of a potential loop. Remove n_2 and move on to n_3. From this point on, an acyclic graph is detected. For each potential loop in the list, we use the Branch and Bound algorithm to find at least one path from entry point to returning point to confirm the existence of a loop. For example, the loop with entry of n_2 and returning of n_9 is confirmed to be a real loop because there is at least one sequence from n_2–n_9. This detected loop is denoted as L_1.

(2) *Identifying simple sequences*. Starting from hot-spots, the sequence search moves both upwards and downwards to reach starting and exiting points. The sequence sets collected from both upwards and downwards traversal are combined into a full-path set using concatenation. For the example in Fig. 2, starting from n_8, backtrack to find the previous nodes that can directly reach the current node. Add these nodes to the collected sequences and repeat this backtracking until it reaches the starting node. When reaching node n_5, we have one sequence of $\langle n_8, n_7, n_5\rangle$, because there is only one previous node before this point. At point n_5, we have two previous nodes, n_3 and n_4. Since n_4 has a higher priority than n_3, we add n_4 to the first sequence and create a new sequence with n_3 added. Now we have two sequences: $\langle n_8, n_7, n_5, n_4\rangle$ and $\langle n_8, n_7, n_5, n_3\rangle$. For each sequence, the backtracking continues toward the starting point. Both sequences reach n_2, which again has two previous nodes, n_1 and n_9. The node n_2 is a looping node as detected by the previous step, so that n_2 is marked as beginning of a loop and n_9 as the returning. At the end of this backtracking, we obtain two sequences: $\langle n_8, n_7, n_5, n_4, n_3, n_2^{L_1}, n_1\rangle$ and $\langle n_8, n_7, n_5, n_3, n_2^{L_1}, n_1\rangle$. Similarly, we can use forward tracking to find all sequences from hot-spots to an exiting point. Starting from n_8, find its next node n_9, and then from n_9 to find n_2. Again n_2 has two next nodes, n_3 and n_{10}. Since n_3 is inside the looping segment marked by n_2 and n_9, it will not be added to the sequence. At the end of this downward searching, we obtain one sequence: $\langle n_8, n_9^{L_1}, n_2, n_{10}\rangle$. Concatenating the sequences from upwards backtracking and downwards searching, we obtain two high priority sequences: $\langle n_1, n_2^{L_1}, n_3, n_4, n_5, n_7, n_8, n_9^{L_1}, n_2, n_{10}\rangle$ and $\langle n_1, n_2^{L_1}, n_3, n_5, n_7, n_8, n_9^{L_1}, n_2, n_{10}\rangle$.

(3) *Deriving patterns*. Using the detected loop notation, the two sequences can be written as one rule $\langle n_1, L_1{*}, n_2, n_{10}\rangle$. We use an enhanced grammar with dominator factor to represent the production rules of node sequence represented by a control flow graph. We use grammar variables to represent loop segment and "*" to represent the fact that we ignore the loop counts at this step. The terminal symbols are listed in a priority descending order, e.g., the first terminal's priority is higher than the second's. The production rule of the two high priority sequences with loops can be denoted as:

$$\begin{aligned} \boldsymbol{P} = \boldsymbol{S} \rightarrow \boldsymbol{S}_1\mathbf{L}_1^*\boldsymbol{S}_3, \text{where } \boldsymbol{S}_1 &= \{s_1\},\\ \mathbf{L}_1 &= \{s_{21}|s_{22}\},\\ \boldsymbol{S}_3 &= \{s_3\}, s_1 = \langle n_1\rangle,\\ s_{21} &= \langle n_2, n_3, n_4, n_5, n_7, n_8, n_9\rangle,\\ s_{22} &= \langle n_2, n_3, n_5, n_7, n_8, n_9\rangle, \text{and}\\ s_3 &= \langle n_2, n_{10}\rangle. \end{aligned}$$

(4) *Generating ordered sequences*. With the node sequence patterns, we can use some heuristics to select sequences for test generation in the next step. The first selection is to pick the alternative sequence elements (terminals) with the highest priority. Since we put terminals in a prioritized order, the generated sequences are naturally given in a prioritized order. For our example, the first sequence generated without loops is $\langle n_1, n_2, n_3, n_4, n_5, n_7, n_8, n_9, n_2, n_{10}\rangle$. The second sequence is $\langle n_1, n_2, n_3, n_5, n_7, n_8, n_9, n_2, n_{10}\rangle$. If all those no-loop sequences are considered, and none of them are satisfiable, then we will detect whether some variables inside the loop are affected by the loop count. If so, we use the loop count as a variable for our constraint solver, which will be explained in the next section. If even this fails, we determine that the current hotspots are not reachable and try to select a second set with the second largest priority values.

The path selection obtains a set of high priority sequences that can be instantiated into actual test cases. It is important to note that this sequence generation algorithm is general, so that it is independent of the hotspot selection criteria. Besides coverage, we can also use other criteria such as change frequency, program complexity, etc., to select hot-spots. The first path in the list to be selected for test case generation is denoted as $\mathbf{P}_{CU}$.

This approach of path selection takes into consideration calling relationship among class methods and program functions. Since the tests going through hot-spots will cover high number of code, including code reached by a function call from the function where the hot-spot resides. This makes the later step of constraint selection and solution much simpler, because it does not longer need to deal with function calls and their return values. Consequently, the test data generation in the next step seemingly deals only with one method. Using program prioritization in path selection to deal with function calls and to simplify test data generation is one of the key innovations of this paper.

3. Our constraint solving method to test generation

After a code sequence is selected on the method with the hot-spot, we use our constraint analysis method to find object attribute and method parameter values for generating tests to traverse through the selected sequence. It includes three steps, constraint derivation, constraint solving, and test case representation, which are explained in the subsequent sections.

3.1. Constraint derivation

We can use either a top–down or a bottom–up approach to derive constraints from selected paths, which are the sequences of selected nodes on a control flow graph. That is, the constraints deriver may start at either end of the selected path when deriving the set of associated constraints. This step includes three sub-steps, constraint collection, feasibility check and redundant constraint removal.

3.1.1. Constraint collection

The constraint collector gathers constraints along the selected path for test generation, so that the generated test execution will traverse the path. It performs the following three tasks:

(1) *Constraint identification*. For each node (**N**) on the path $\mathbf{P}_{CU}$ (CU is the highest priority code unit, hot-spot), the constraints collector determines the constraints associated with the node **N** such that when these constraints are satisfied, then an execution of the software code (or its translated code) will continue to the next node on the path $\mathbf{P}_{CU}$. For loops, if the loop execution count is not yet a variable, a new variable will be created for it. For example, the index of a FOR statement is a loop count variable. In many cases, a loop count variable needs to be created for WHILE loops. In the previous example, a count variable l_1 can be created for the loop from n_2–n_9 and l_1 can participate in a constraint on an edge from one node to the other.

Again, note that the constraint collector may use either a top–down or a bottom–up approach to derive constraints from the path $\mathbf{P}_{CU}$. That is, the constraints collector may start at either end of the path $\mathbf{P}_{CU}$ when deriving the set of associated constraints. For example, consider a Java program as given in Fig. 3(a), its bytecode in Fig. 3(b) and its control flow diagram in Fig. 3(c). Each node in the graph is denoted as the beginning line number of the corresponding bytecode. For example, node 8 includes lines 8 and 9, and node 12 includes lines 12 and 14. The structural and overall call dependability analysis finds node 27 to be the highest priority for improving the coverage. The calculation of program unit priority is given in [2]. If the path $\mathbf{P}_{CU}$ going through the control flow graph of Fig. 3(c) is the node sequence of $\langle n_0, n_{12}, n_{27}, n_{32}, n_{37}, n_{50}\rangle$, wherein **CU** is, e.g., the code line 27 of Fig. 3(b), then the bottom–up collection of constraints entails performing the following tasks (i) through (v):

(i) Starting from the last node (e.g., node 50), move backwards to node 37. The corresponding constraint for "*iload %9; ifeq #50*" is "$x9 == 0$".
(ii) Similarly, from node 37 to node 32, the constraint "$x8 == 0$" is obtained.
(iii) From node 32 to 27, the constraint "$x7 == 0$" is obtained.
(iv) From node 27 to 12, the constraint "$x4 ! = 0$" is obtained.
(v) From node 12 to 0, the constraint "$x1 ! = 0$" is obtained.

Thus, at the end of the constraint collection for the path $\mathbf{P}_{CU}$, the set of constraints is collected as follows: $x1 ! = 0$; $x4 ! = 0$; $x7 == 0$; $x8 == 0$; and $x9 == 0$.

(2) *Constraint reduction*. For each constraint identified in task (1) immediately above, reduce the constraint through substitutions so that the constraints are only in terms of the following: (i) basic data types (e.g., integer, real, character, string, Boolean, etc.); (ii) input parameters to the path $\mathbf{P}_{CU}$; (iii) fields of input parameters to the path $\mathbf{P}_{CU}$; and (iv) expressions using only (i) through (iii) immediately above together with constants. For example, assume the software code as given in Fig. 4(a)) is input to the constraint collection step and no translation is performed:

Further assume that the path $\mathbf{P}_{CU}$ through the control flow graph corresponding to the code "prog" in Fig. 4(a) is shown in Fig. 4(b) with the path $\mathbf{P}_{CU}$ corresponding to the nodes n_1, n_3 and n_4 together with the directed edges there between. Then there is a single "constraint" to be satisfied for ensuring that the path $\mathbf{P}_{CU}$ is performed, i.e., "$x > 0$". However, the reduction of this constraint requires that "x" in the constraint be replaced by "$x + 1$" so that the corresponding reduced constraint is in terms of the input parameter as required by (ii) immediately above.

a

```
public void myCSP1(boolean x1, boolean x2, boolean
x3, boolean x4, boolean x5, boolean x6, boolean x7,
boolean x8, boolean x9) {
        if ((x1 || x2 || x3) &&
           (x4 || x5 || x6) &&
           (x7 || x8 || x9))
        System.out.println("Hello CSP1");
}
```

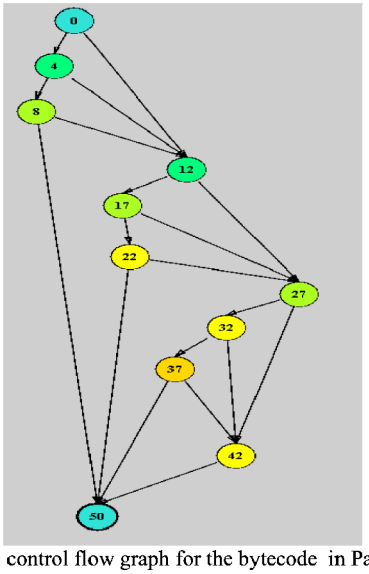
A sample Java source code

b

```
void myCSP1(ZZZZZZZZZ)V
  0:  iload_1
  1:  ifne        #12
  4:  iload_2
  5:  ifne        #12
  8:  iload_3
  9:  ifeq        #50
 12:  iload               %4
 14:  ifne        #27
 17:  iload               %5
 19:  ifne        #27
 22:  iload               %6
 24:  ifeq        #50
 27:  iload               %7
 29:  ifne        #42
 32:  iload               %8
 34:  ifne        #42
 37:  iload               %9
 39:  ifeq        #50
 42:  getstatic           java.lang.System.out
Ljava/io/PrintStream; (2)
 45:  ldc         "Hello CSP" (50)
 47:  invokevirtual       java.io.PrintStream.println
(Ljava/lang/String;)V (4)
 50:  return
```

The corresponding Java bytecode of sample Java code in Part(a)

c



The control flow graph for the bytecode in Part (b)

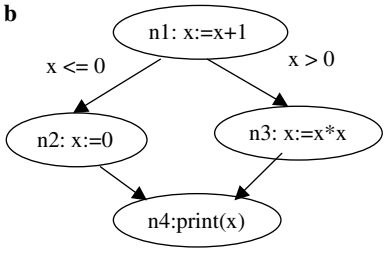
Fig. 3. A sample Java source code with Boolean expressions and its corresponding bytecode and control flow graph.

a

```
prog(x) {
   int x;
   x := x +1;
   if (x > 0) then
      x := x * x;
   else
      x := 0;
   print(x).
}
```

A small program sample to illustrate constraint reduction algorithm

b



The control flow graph of sample program at (a)

Fig. 4. A constraint reduction example.

Thus, the resulting reduced constraint set is $\{x+1>0\}$. Accordingly, the term "constraint" as used in the description of the constraint analysis is more general in that assignments may be considered constraints when such assignments provide necessary information for reducing another constraint along the path $\mathbf{P_{CU}}$, wherein the constraint identifies an edge of the path $\mathbf{P_{CU}}$.

In performing constraint reductions, complex records or objects may need to be parsed so that if values are determined for their (basic data type) fields which make the path $\mathbf{P_{CU}}$ feasible, then an appropriate instance of the complex record/object can be subsequently generated (explained in subsequent steps) for testing the software code.

(3) *Constraint enlistment*. Add each reduced constraint to a list of constraints (referred to herein as **CONSTRAINTS_LIST$_{CU}$**). Before proceeding to the next step of constraint feasibility checking, it is worthwhile to provide a further example constraint derivation on a more complex software code; i.e., software code that includes a string comparison and an object function call. The Java code of Fig. 5(a) is illustrative of this more complex software code, wherein Fig. 5(b) shows the corresponding bytecode that can be output by the translator, Due to space constraint, only some key lines of bytecode are shown.

To select a path as $\mathbf{P_{CU}}$, a representation of the control flow graph corresponding to the bytecode of Fig. 5(b) must be generated. Fig. 5(c) shows such a control flow graph, wherein each node is labeled with the number of the code line in Fig. 5(b) that starts the sequence of code lines identified by the node. Assuming the selected path for $\mathbf{P_{CU}}$ includes the node sequence: $\langle n_0, n_{25}, n_{48}, n_{82}, n_{105}\rangle$, the forward (i.e., top–down) constraint derivation for this example entails performing the following tasks (1) through (4):

a

```
public void myexample2 (String msg1, String msg2) {
        id = ((Integer)hashmap.get(msg1)).intValue();
        if (id > 0) {
            if (msg1.equals(msg2)) {
                System.out.println("The lengths are equal. ");
                test2Print(msg1);
            } else {
                System.out.println("The lengths not equal");
                if (msg1.length() > msg2.length()) {
                    System.out.println("msg1 is longer");
                    samplePrint(msg1);
                } else {
                    System.out.println("msg2 is longer");
                    samplePrint(msg2);
                }
            }
        } else
            System.out.println("wrong id");
}
```

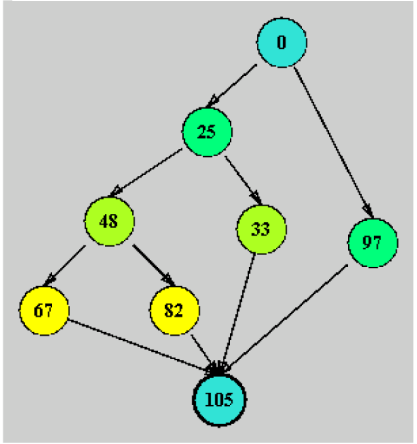
Another sample Java source code

b

```
void myexample2(Ljava/lang/String;Ljava/lang/String;)V
0:   aload_0
1:   ***
22:  ifle          #97
25:  aload_1
26:  ***
30:  ifeq          #48
33:  getstatic java.lang.System.out Ljava/io/PrintStream; (2)
36:  ***
45:  goto          #105
48:  getstatic java.lang.System.out Ljava/io/PrintStream; (2)
51:  ***
64:  if_icmple          #82
67:  getstatic java.lang.System.out Ljava/io/PrintStream; (2)
70:  ***
79:  goto          #105
82:  getstatic java.lang.System.out Ljava/io/PrintStream; (2)
85:  ***
94:  goto          #105
97:  getstatic java.lang.System.out Ljava/io/PrintStream; (2)
100: ***
105: return
```

The corresponding Java bytecode of Part(a)

c



Part(c): The control flow graph for the bytecode in Fig 5(b)

Fig. 5. Another sample Java source code and the corresponding bytecode with string type constraints.

(1) From node 0 (i.e., code lines 0–24), the following assignment and a condition are identified as constraints (all expressions involving "equal", "not-equal", "larger" "less", "large equal" and "less equal" are identified as constraints):
Sample.id = Sample.hashmap.get(aload_1) and
Sample.id > 0.
Note that although the assignment identified here is not a constraint in the usual sense of the word, such assignments are considered constraints herein as discussed above.
(2) From node 25, the following string comparison condition is identified as a constraint: "*String.equals(aload_1, aload_2)*".
(3) From node 48, the following integer comparison condition is identified as a constraint: "*length(aload_1) length(aload_2)*".
(4) Nodes 82 and 105 will then be checked for constraints, but no constraints will be identified since there is no comparison operators associated with these nodes.

Overall the forward traversal of the selected path generates the following constraints:

(1) *Sample.id = Sample.hashmap.get(aload_1)*
(2) *Sample.id > 0*
(3) *String.equals(aload_1, aload_2)*
(4) *length(aload_1) <= length(aload_2)*

The object function call involved still needs to be considered in this case because it is related to one of the basic operations, "equal". Other function calls, such as "test2Print" and "samplePrint", do not need to be considered in the constraint collections because their coverage is already considered during hot-spot and path selection, as discussed previously [2].

Returning now to Fig. 1, a determination should be made as to whether the constraints on **CONSTRAINTS_LIST$_{CU}$** are consistent, thus indicating whether or not the path **P$_{CU}$** is feasible. If not, then a step is performed for determining whether there is another path through the control/data flow graph that goes through a node, wherein the node identifies the code unit **CU**. If not, then another step is again performed, wherein the next (if any) highest priority code unit is selected and assigned to **CU** for again performing path selection. Note, however, if there is no further code units to be assigned to **CU**, then the test generation framework is exited. Alternatively, if the result indicates that there is an additional path through the control/data flow graph that goes through a node identifying the code unit **CU**, then this step and steps following are performed again.

3.1.2. Feasibility check

In determining whether the constraints on **CONSTRAINTS_LIST$_{CU}$** are consistent (equivalently, that path **P$_{CU}$** is feasible), a novel evaluation method is used

to decide whether constraints have conflicts (and accordingly not consistent). An example illustrates this novel constraint evaluation method. Assume that there are two constraints, "$x > 7$" and "$x < 6$". Two expressions are generated from these constraints. That is the first constraint ("$x > 7$") is represented as x belongs to $[7 + e,\ MAX\text{-}X\text{-}TYPE]$, where e is the smallest positive number of variable x's data type, and *MAX-X-TYPE* is the maximum value of x's data type. For example, assuming that x is of integer data type, then $e = 1$. *MAX-X-TYPE* can be determined similarly as the largest possible integer that is representable by an integer data type. Note that *MAX-X-TYPE* maybe computer-dependent. Moreover, for some data types such as real, e may be computer-dependent as well. Accordingly, in one embodiment, for data types such as integer and REAL, values for e and *MAX-X-TYPE* may be determined that are realizable in most computers, and additionally are respectively small enough and large enough so that a range such as $[7 + e,\ MAX\text{-}X\text{-}TYPE]$ will include substantially all the computer representable solutions regardless of the computer. Thus, regarding the expression $[7 + e,\ MAX\text{-}X\text{-}TYPE]$, the lower bound of the range is 8. So by replacing the variable x with this lower range, and the above original two constraints become "$8 > 7$" and "$8 < 6$", and the following expression "$(8 > 7)$ & & $(8 < 6)$" can be generated and then evaluated. Similarly, *MAX-X-TYPE* will clearly be larger than 6. Thus, due to the linearity of the constraints, all possible evaluations are determined to be false, and thus it is concluded that the original constraints are not consistent.

For non-linear constraints, a value of each constraint variable can also be determined by determining a set of lower and upper bounds in a manner similar to that described immediately above. For example, suppose for a given path ***P*** (of the appropriate control/data flow graph), there are exactly the two constraints "$X^2 > 9$" and "$X < 3$" for determining feasibility of the path. The first constraint yields two segments, $[3 + e,\ MAX\text{-}X\text{-}TYPE]$ and $[MIN\text{-}X\text{-}TYPE,\ -3 - \mathrm{e}]$. Since both "$X = MIN\text{-}X\text{-}TYPE$" and "$X = -3 - \mathrm{e}$" satisfies the two constraints, "$X^2 > 9$" and "$X < 3$", it can be concluded that the path ***P*** is feasible.

For some collections of constraints, various types of searches may be used for identifying whether the constraints are consistent. In particular, a binary search may be used. For example, suppose for a given path ***P*** (of the appropriate control/data flow graph), there are exactly the two constraints "$X^2 > 9$" and "$X < -9$" and "$\mathrm{X} > -4$" for determining feasibility of the path. Starting with the variable range of $[MIN\text{-}X\text{-}TYPE,\ -3 - \mathrm{e}]$ corresponding to the first constraint, the boundary checking fails (i.e., $MIN\text{-}X\text{-}TYPE < -4$, and, $-3 - \mathrm{e} > -9$). Accordingly, the range $[MIN\text{-}X\text{-}TYPE,\ -3 - \mathrm{e}]$ is decomposed into $[MIN\text{-}X\text{-}TYPE,\ (MIN\_X\_TYPE\text{-}3 - e)/2]$ and $[(MIN\_X\_TYPE - 3 - \mathrm{e})/2,\ -3 - \mathrm{e}]$, and the end points of these ranges are tested for consistency. It turns out that $[MIN\text{-}X\text{-}TYPE,\ (MIN\_X\_TYPE - 3 - \mathrm{e})/2]$ is a feasible solution.

The following tasks may be used for determining the feasibility of a given path ***P*** after all constraints for the path ***P*** have been reduced: suppose we have a constraint set of ***C*** and a set of identifiers ***V***.

(1) First, find all the constraints that involve only one single identifier and put them in a set $\boldsymbol{SI} = \{C_1, C_2 \ldots C_{\mathrm{n}}\}$.
(2) For each constraint in ***SI***, find its corresponding identifiers and the set ***SI*** becomes $\boldsymbol{SI} = \{C_1\ (V_{c1}), C_2(V_{c2}) \ldots C_{\mathrm{n}}(V_{cn})\}$.
(3) For each constraint in ***SI***, determine a range for the identifier in the manner of the examples above. At the end of this step, we obtain a set of ranges $\boldsymbol{R} = \{R(V_{c1}), R(V_{c2}) \ldots R(V_{cn})\}$.
(4) Find all simple "equality" comparison constraints with only one variable at each side of the equation, such as $V_i = V_j$ and $V_j = V_k$, and put them in a set $\boldsymbol{E} = \{e_1, e_2 \ldots e_x\}$, where $e_1.left$ is the identifier on the left and $e_1.right$ is the identifier on the right. Use ***E*** as the input and a new empty set ***F*** to store outputs, run the algorithm as given in Fig. 6 (in the algorithm, == means identifier name equal). The algorithm replaces variables of the same value with one single variable name. The output of the algorithm is a new set ***F*** with simple "equality" comparison constraints.
(5) For all elements in $F = \{e_1 \ldots e_y\}$ and all elements in R, replace the identifiers in R with the right-side identifiers in F. For example, suppose $e1.left$ is the same identifier as V_{c1}, replace V_{c1} with $e1.right$. We obtain a new set of $\boldsymbol{R} = \{R(V_{c1}), R(V_{c2}) \ldots R(V_{cn})\}$.
(6) Determine a range for each variable $V_{c1}, V_{c2} \ldots V_{cn}$. If some of them are the same identifier, find the intersection of their ranges. If the intersection is empty, no feasible value can be found for the identifier and the constraints are determined to be infeasible and quit. For example, if V_{ci} is the same identifier as V_{cj}, the intersection of V_{ci}'s range L_{ci} and V_{cj}'s range L_{cj} is the range for the identifier V_{ci} and V_{cj} can be removed. If the range is empty, infeasibility is detected for this set of constraints. Move on to the next set of constraints. If there is no more constraint sets for evaluation, the path is determined to be infeasible and this process ends.
(7) After the above intersection operation, V_{c1}, $V_{c2} \ldots V_{cnew}$ are all different identifiers. For each of these identifiers, find their range boundaries. Since the range can include multiple segments, the number of boundary values can be more than 2.
(8) Use the boundary values to replace the identifiers in the rest of constraints that involve more than one identifier and obtain a new set of constraints. Repeat this process by starting at step (a), until no more single identifier constraints can be found.

```
equality-replace(in: E, out: F)
{
    While not-empty(E) do {
        e1 = first-element(E);
        removeFromOneSetAndAddToTheOther(e1, E, F); // remove e1 from E and add it to F
        for (int i= 1; i <= length(E); i++) {
            if  ((ei.left == e1.left) || (ei.left == e1.right)) {
                ei.left = ei.right;
                ei.right = e1.right;
                removeFromOneSetAndAddToTheOther(ei, E, F);
                continue;
            }
            if ((ei.right == e1.left) || (ei.right == e1.right)) {
                ei.right = e1.right;
                removeFromOneSetAndAddToTheOther(ei, E, F);
            }
        }
    }
}
```

Fig. 6. The equality replacement algorithm for feasibility checking.

3.1.3. Redundancy removal

In addition to checking the feasibility of various paths through the software code (equivalently, the corresponding control/data flow graph), redundant constraints can also be removed. For example, the following four constraints are obtained from the Java bytecode program displayed in Fig. 5(b). To enhance the readability, variables/identifiers are in bytecode format, the conditions are in simpler format, "=" is "*eq*", ">" is "*lg*", etc.

(1) *Sample.id = Sample.hashmap.get(aload_1)*
(2) *Sample.id > 0*
(3) *String.equals(aload_1, aload_2)*
(4) *length(aload_1) <= length(aload_2)*

Note that constraint (3) immediately above implies that "*length(aload_1) == length(aload_2)*" which is a subset of constraint (4) immediately above. Therefore constraint (4) is redundant and can be removed from the constraint list. Also, constraints *(1)* and *(2)* can be combined into "*Sample.hashmap.get(aload_1) > 0*". Thus, the following steps may used for removing redundant constraints in a constraint list associated with a path through a control/data flow graph.

(1) Replace variable identifier with its assigned value. For example replace item (2) *Sample.id* with *Sample.hashmap.get(aload_1)* on item (1).
(2) For all identifier "equals", all their methods or sub-fields are equal. For example, "$X = Y$", and "$F(X) = F(Y)$" are overlapped, and only "$X = Y$" is needed, and the same goes for "$X = Y$" and "$F(X) >= F(Y)$".

Considering the code in Fig. 5(a) and (b), removal of redundant constraints results in only two constraints remaining for the Java bytecode in Fig. 5(b). After applying the two steps mentioned above to the original four constraints, we came to the two constraints below:

(1) *Sample.hashmap.get(aload_1) > 0*, and
(2) *String.equals(aload_1, aload_2)*.

Referring again to Fig. 1, if the path $\mathbf{P_{CU}}$ is determined to be feasible, then move on to the next step, wherein the constraints on $\mathbf{CONSTRAINTS\_LIST_{CU}}$ are solved via the constraint solver. For each identifier instanced in one of the constraints, there is at least one collection of ranges for these identifiers such that a selection of a value from the corresponding range for each identifier will cause the path $\mathbf{P_{CU}}$ to be traversed. Accordingly, random values within the corresponding ranges for each of the identifiers may be selected to obtain a set of values for generating test code. Thus, one or more such sets may be generated in this manner. Subsequently, test code is generated from each of the sets. In particular, the code for appropriately initializing any complex records/objects is generated. The details of this step will be explained in the next subsection.

3.2. Constraint solving

The goal of the constraint solver is to check whether the path is feasible and, if so, what instances of data values should be used as parameter and object constructor inputs. Most constraint solvers handle only numerical constraints involving basic data types such as integer, float, and Boolean. None of them handles composite types such as String and Objects. Our constraint solver can handle both. It includes the following two tasks:

(1) Determining the ranges of object constructor and method parameter variables, and
(2) Generating the exact values of method and constructer parameters.

3.2.1. Solving variable ranges

This task finds the ranges of variables that satisfy all the constraints. With numerical types, we can use a binary search method to deduce the variable range. For example, suppose we have two constraints as "$X^2 > 9$" and "$X - 9$". We start with the variable range of [*MIN-X-TYPE*, $-3 - e$]. If the boundary checking fails, we will next try

[*MIN-X-TYPE, (MIN_X_TYPE*−3 − e)/2] and [*(MIN_X_TYPE*−3 − e)/2, −3 − e]. It turns out that [*MIN-X-TYPE, (MIN_X_TYPE*−3 − e)/2] is a feasible solution.

The goal of constraint solving for test data generation is to find *any* data values that satisfies all the constraints. It does not need to find *all* possible data values. Therefore, for the above example, once a feasible range is determined, the constraint solving can be stopped and the one solution can be used to generate a test case.

For the two constraints of the Java program in Fig. 5(a), the object "*hashmap*" needs to be created. It includes two fields, *index* and *value*. We use the leaf variables to replace objects in the constraints and obtain the new constraints as "*(hashmap1.index == aload_1) && (hashmap1.value > 0) && (aload_1 == aload_2)*".

The "String" data type is also treated as a dynamic object. Its first field is the length of the string and the number of the fields is the same as the length of the string. Each of the subsequent fields has the basicType of characters. The values of "*String*" type variables mostly fall into the following two categories:

(1) It is a constant value defined in the program, in a file or in a database. For example, the possible values for the variable "*loginID*" are most likely to be stored in a database; therefore, the variable range is the list of "*LoginID*" inside the database.
(2) It is a string value input from users. The variable range of such a "String" type is determined by both its length and its content. For example, if a "*String*" variable is not involved in any constraints, then its length range is from 0 to the maximum value of integer and each of its character can be any legitimate characters.

When all variable ranges are determined, an instant of the variable values must be selected to generate individual test cases, as described in the next section.

3.2.2. Parameter instantiation

With the variable ranges obtained from the constraint solver, we can decide on actual variable values with the following approaches:

1. Use a random number generator to find a value within the given range. For example, for the range of "*hashmap.value > 0*", we can use a random number generator to get a value of say 3452.
2. Provide the range information for users to select an actual variable value. Again for the same constraint, the user may select a more commonly used value of say 1.
3. Use past experience and heuristics to find an actual value. For example, we can use the heuristic that the value is most often to be around 10 during field usage. Using this field usage criterion, a value of 10 is selected.

3.2.3. Handling composite objects

Our control flow graph is generated based on Java bytecode. Various data types represented in the lower level program language (such as bytecode), and/or the set of operations that can be performed in the lower level programming language is reduced. For example, Boolean operations of Java code, such as "||" and "&&", may be translated into branching instructions in the lower level language of bytecode. Therefore, our solver does not need to deal with these Boolean operations. Accordingly, the constraints derived by this step are in general simpler than if the corresponding constraints that would be generated directly from Java code. However, there is a tradeoff in that the number of constraints generated increases. Such a tradeoff is believed worthwhile in that the software for solving these simple constraints is not as complex. Consider again the example given in Fig. 3; the Boolean operations in Fig. 3(a) have been translated into branching operations in Fig. 3(b). For example, the expression "*x1||x2||x3*" in Fig. 3(a) corresponds to Line 0–9 in Fig. 3(b). The operands such as "&&" and "||" are translated into branching of new nodes. For example, node 0, 4 and 8 represents the "or" operation of "*x1 || x2 || x3*". Accordingly, in the example, the only operators and operands left from which constraints can be generated are those related to arithmetic operations such as greater than, less than, and equal to.

For solving constraints with complex object types, the objects need to be decomposed into a tree structure with basic data types as its leaves. Solve the leaf variables and then recompose them back into an object. Fig. 7 shows an example of handling a composite object. Suppose an object "objectTop" is an instant of a class that has two attributes "object1" and "basicType1", and the class inherits from another class "objectSuper" also with two attributes "objectS" and "basicTypeS". The "objectS" is again a composite type and the "basicTypeS" is a variable of basic type. The decomposition and re-composition work as follows. The object "objectTop" is first decomposed into four fields, two of its own and the other two from its parent class. Similarly, the objects "object1" and "objectS" are further decomposed into its attributes plus its parents' attributes. Repeat the above procedure, until all leaves on the decomposition tree are variables of basic types. After obtaining values from each leave variable, the re-composition of the

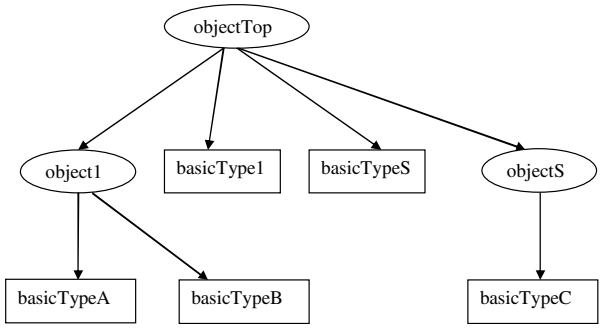


Fig. 7. An object decomposition diagram.

"objectTop" starts from the bottom of the tree. One important point is that the values of the super classes should also be set if they are involved in constraints of the path.

In summary, at the end of the constraint solving, one value of each basic variable is selected for actual test case construction and all dynamic objects are created with their appropriate fields set.

3.3. Test case representation

This task represents tests and test data in the same programming language as the original code, so that the generated test cases are directly compilable and executable alongside the original code. The actual test cases should set up the testing by initializing class attributes and method parameters. It should also tear down the test setting when it is done. Since we are using an existing unit-testing framework, we assume that we have three methods for the test framework, i.e., setup, run, and tear-down. The "run" method calls the method under test, using the generated test data as parameters.

One common situation in complex software is that the variables of class attributes and method parameters can be dynamic object types, as discussed previously. The variables for the objects should also be instantiated. For example, if we have *hashmap1.index* = "*ABC*" and *hashmap1.value* = 10, we can construct and instantiate the object hashmap as follows:

hashmap1 = new Hashmap();
hashmap1.put("ABC", 10);

At the end of this step a test case given in the original programming language is generated. Also, the actual variable values of basic types are instantiated. Fig. 8 shows an example of generated test case. Unlike other program code generator, the generated test case code is highly readable, which makes it easier for users to augment or modify to get more test cases. The generated test cases provide a better framework for users than the original one with stubs and mocks.

```
Class class-under-test {
     int attibute1;
     class A attribute2;
     class-under-test(attribute1, attribute2);
     void method-under-test(int p1, class B p2)
     { line1; line2  }
}
class A { int attributeA1, }
class B { int attributeB1, }

The generated framework from the above class should look like:

class example-framework extends test-frame-work {
     int attribute1;
     int  p1;
     class A objectA;
     class B objectB;

     void setup() {
             attribute1 = a;
             p1 = b;
             int attributeA1 = c;
             objectA = new A(attributeA1);
             int attributeB1 = d;
             objectB = new B(attributeB1);

}

void run();
{ setup();
   testclass = new
class-under-test(attribute1, objectA);
   testclass.method-under-test(p1, objectB);
    teardown();
}
}
```

Fig. 8. A sample automated generated test case.

3.4. Implementation and observations

We implemented the constraint solver and automatic test generation in an automatic testing tool suite. It automates the entire process of unit testing, including program analysis, test generation, test execution, and debugging. The constraint solver derives constraints from Java bytecode, solves the constraints and generates test case in Java. The solver was implemented in Java.

The constraint solver is very efficient. Table 1 shows the time of each constraint solving step in handling various programs under test.

Table 2 gives a summary of the programs on which we have tried the test generator. As it can be seen, automatic test generation contributes to the improvement of codecoverage and discovery of defects.

Generated test cases can be executed and the test results are fed into the eXVantage tool automatically. Based on the traces collected during testing, the tool suite can generate coverage reports and localize performance bottlenecks and behavioral bugs.

Our experiments show the constraint solver is able to resolve most constraints that we have encountered. The only one situation that it fails is handling of abstract data types as method parameters. Investigation of a solution is in progress. Nevertheless, the current version of our constraint solver is still ahead of other existing constraint solvers in handling dynamic date types. Of the 10 Java packages that the test generator takes as inputs, the automatic generated test cases achieve a coverage ranging from 20% to 98%. The low coverage ones are due to the complexity of the constraints that our handler cannot collect and resolve. We also observed that better programs with lower complexity metrics can achieve higher codecoverage with automatic test generation. More details of our experimental results will be given in a future report.

4. Related work

Edvardson [12] presented a survey on program-based automatic test data generation where a test data generator system is divided into three parts: program analyzer, path selector, and test data generator. The program analyzer analyzes the source code of the program and gives inputs to the remaining parts of the system. The path selector goes through the program data provided and selects a set of paths which lead to high coverage with respect to certain criteria, such as statement coverage, branch coverage, path coverage (which requires the traversal of each path in the flow graph), etc. The test data generator takes the selected paths as arguments and derives input values that exercise the given paths. However, most of such work is based on toy programs, i.e., programs that are either very short in

Table 1
Time used for each step of test generation

Path length (line of code)	Constraint derivation (ms)	Constraint solving (ms)	Test generation (ms)
28	133	363	27
67	1711	4786	64
96	2043	6712	76

Table 2
Automatic generated tests can find bugs and reduce testing time

Size	Bugs found	Test coverage improvement	Automatic test generation and execution time
2M	12	2 times and more	2 weeks
50K	1	10%	2 h

length or low in complexity or that lack the use of many standard language features. This is very different from our approach, as our technique will be applied to large, complex telecommunication systems developed at Avaya.

Korel [13] indicated that most of the existing test data generators [14–18] use symbolic evaluation to derive test data, which may require complex algebraic manipulations, especially in the presence of arrays. Instead, he proposed a dynamic approach based on the actual execution of the program being tested, function minimization methods, and dynamic data flow analysis. For each path in the program, this approach decomposes the program into a sequence of sub-goals. Then, a solution for each sub-goal can be obtained by using its minimization function. The basic search strategy for a minimization function is to repeatedly change the value of one input variable at one time until the solution is found. Two search methods can be used in this strategy. One is the exploratory search, which tries to find a direction for future search by slender altering of the value of only one variable repeatedly. If a search direction cannot be determined by using this method, pattern search can be applied to shorten the search process by large moves of the variable's value in that direction. Also, a heuristic approach, based on dynamic data flow analysis, can be used to further reduce the search effort in the exploratory search and pattern search methods. This heuristic approach only considers the input variables that have an influence on a particular minimization function, thereby reducing unnecessary effort on other variables. The major problems with this approach are that the programs being tested must be written in a Pascal-like language and the branch predicates have to be very simple (for example, it is assumed that predicates do not contain AND's, OR's, or other Boolean operators).

Micheal et al. [20] reported two experiments on test data generation. One uses random test data generation, where inputs are selected at random in the hope that they exercise the desired software features. This approach tends to fail when the complexity of software increases and when there is a need for test cases to satisfy very complex constraints of the program. The other uses a combinatorial optimization technique. Based on the data collected from their case study, the second approach performs better when software is complex and when the test cases have to satisfy complicated constraints. Micheal et al. [19] also developed a software test generation tool, GADGET, based on combinatorial optimization. GADGET uses five different techniques (namely, gradient descent algorithm, simulated annealing algorithm, standard genetic algorithm, differential genetic algorithm, and random approach) to generate test data for C/C++ programs. They concluded that although random testing works well for simple programs, its performance worsens as the complexity of a program increases. They also observed that different test coverage criteria may be satisfied simultaneously. That is, test data generated for one coverage criterion may also be used to satisfy another coverage criterion. Of the five aforementioned techniques, the standard genetic algorithm and simulated annealing algorithm perform better than others in their experiments.

Sy and Deville [21] presented an approach for automated test data generation of imperative programs with only integer, Boolean, and/or float variables. Their objective is to generate test data to satisfy the statement, branch, and path coverage criteria. This approach is based on consistency techniques integrating integer and float variables. In another paper, the same authors reported another study by extending their previous work to programs with procedure calls and arrays. A major difference between these two studies is that in the first one a program under test is represented by an inter-procedural control flow graph, whereas in the second one a program is transformed into a static single assignment (SSA) form. The major concern with these approaches is that they can only be applied to C programs with very limited features. More specifically, only integer inputs can be handled. As for procedure calls, only the pass-by-value mechanism for passing parameters is considered.

Zeng et al. [22] suggested that the functional test generation problem guided by constraints such as reaching a particular state of the design, exercising a branch, or covering a piece of code can be posed as a satisfiability problem (SAT). The main objective of [22] is to present an approach for solving SAT (satisfiability problem) based on the constraint logic programming (CLP) technique. However, this satisfiability solver (CLP-SAT) based on Gprolog has a limitation on the integer domain as the Gprolog solver can currently support integers up to the range 2^{28}.

5. Conclusion and future directions

This paper presents a framework for automatic test data generation. It first selects a program path based on program priority analysis. The focus of this paper is using constraint solvers for automatic test data generation from selected paths. It includes two major advancements as compared to other existing work in this area: (1) Java bytecode-based constraint derivation, and (2) handling of dynamic composite objects. We implemented the solver in an automatic test generation tool suite. Experimental results show that the solver can handle most constraints encountered in actual industrial software.

The execution of the generated test case provides a feedback to selecting the next path for test generation. The framework re-calculates the priority of each node in the control flow diagram of the program and selects a new path based on new priorities. The framework repeats this process until no more paths can be found or permitted time runs out. In the steps proceeding the constraint solving and test generation, we claimed two major contributions: solving functional-call class dependency issue through priority calculation and effective path selection taking into consideration program priority and calling relationship among class methods.

One situation our constraint solver cannot handle is abstract data types used as parameters. Our future research direction is to extend the constraint solver to have even more difficult situation, such as abstract types, and larger constraint sets from more complex program structures. We hope our work will contribute to automatic test generation in general.

Another limitation of the method presented in this paper is test result validation. The generated tests do not include test oracle part of the code. In unit testing framework, testing oracles are presented in assertions. One of our future research directions is to generate also assertions automatically.

References

[1] H. Agrawal, J.J. Li, W.E. Wong, et al., Mining system tests to aid software maintenance, IEEE Comput. 31 (7) (1998) 64–73.

[2] J. Jenny Li, Prioritize code for testing to improve code coverage of complex software, in: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), October 2005, Chicago, IL.

[3] Sami Beydeda, Volker Gruhn, BINTEST – Binary search-based test case generation. Computer Software and Applications Conference (COMPSAC03), 03.11.2003–06.11.2003, Dallas, USA.

[4] L. Bromstrup, D. Hogrefe, TESDL: experience with generating test cases from SDL specifications, Proceedings of the Fourth SDL Forum, Elsevier, Amsterdam, 1989, pp. 267–279.

[5] J. Offutt, A. Abdurazik, Generating tests from UML specifications, Proceedings of International Conference on the Unified Modeling Language, October 1999, Fort Collins, CO, pp. 416–429.

[6] J. Hartmann, C. Imoberdorf, M. Meisinger, uml-based integration testing, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), August 2000, Portland, Oregon, USA, , pp. 60–70.

[7] W.T. Tsai, R. Paul, Z. Cao, B. Xiao, L. Yu, Adaptive scenario-based testing using UML, in: OMG's Third Workshop on UML for Enterprise Applications: Model Driven Solutions for the Enterprise, October 2002, San Francisco, CA, USA.

[8] J.J. Li, W.E. Wong, Automatic test generation from communicating extended finite state machine (cefsm)-based models, Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), April 29–May 1, 2002, Washington, D.C.

[9] A. Petrenko, N. Yevtushenko, R. Dssouli, Testing strategies for communicating FSMs, Proceedings of the Seventh International Workshop on Protocol Test Systems (IWPTS), November 1994, Tokyo, Japan, pp. 193–208.

[10] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The ATEG system: an approach to testing based on combinatorial sesign, IEEE Transactions on Software Engineering 23 (27) (1997) 437–444.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., The MIT Press, USA.

[12] J. Edvardson, A survey on automatic test data generation, Proceedings of Second Conference on Computer Science and Engineering, October 1999, Linköping, Sweden, pp. 21–28.

[13] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.

[14] N. Gupta, A.P. Mathur, M.L. Soffa, Generating test data for branch coverage, in: Proceedings of Automated Software Engineering, 2000, pp. 219–222.

[15] C. Ramamoorthy, S. Ho, W. Chen, On the automated generation of program test data, IEEE Transactions on Software Engineering SE-2 (4) (1976) 293–300.

[16] L. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions Software Engineering 2 (3) (1976) 215–222.

[17] W. Howden, Symbolic testing and the DISSECT symbolic evaluation system, IEEE Transactions on Software Engineering 4 (4) (1977) 266–278.

[18] A. Mayrhauser, M. Scheetz, E. Dahlman, Generating goal-oriented test cases, in: Proceedings of the 23rd Annual International Computer Software and Applications Conference (COMPSAC), October 1999, Phoenix, Arizona, USA, pp. 110–115.

[19] C.C. Micheal, G. McGraw, Automated software test data generation for complex programs, in: Proceedings of 13th IEEE International Conference on Automated Software Engineering, October 1998, Honolulu, Hawaii, USA, pp. 136–146.

[20] C.C. Micheal, G.E. McGraw, M.A. Schatz, C.C. Walton, Genetic algorithms for dynamic test data generation, in: Proceedings of the 12th IEEE International Conference, November 1997, Tahoe, Nevada, pp. 307–308.

[21] N.T. Sy and Y. Deville, Automatic test data generation for programs with integer and float variables, in: Proceedings of the 16th IEEE Annual International Conference on Automated Software Engineering (ASE), November 2001, San Diego, California, pp. 13–21.

[22] Z. Zeng, M. Ciesielski, B. Rouzeyre, Functional test generation using constraint logic programming, in: Proceedings of International conference on Very Large Scale Integration (VLSI-SOC), December 2001, Montepellier, Le Corum, France, pp. 375–387.