Test Data Generation based on Binary Search for Class-level Testing*

Sami Beydeda, Volker Gruhn
University of Leipzig
Faculty of Mathematics and Computer Science
Department of Computer Science
Applied Telematics / e-Business
Klostergasse 3
04109 Leipzig, Germany
{sami.beydeda, volker.gruhn}@informatik.uni-leipzig.de

Abstract

One of the important tasks during software testing is the generation of appropriate test data. Various techniques have been proposed to automate this task. The techniques available, however, often have problems limiting their use. In the case of dynamic test data generation techniques, a frequent problem is that a large number of iterations might be necessary to obtain test data. This article proposes a novel technique for automated test data generation based on binary search. Binary search conducts searching tasks in logarithmic time, as long as its assumptions are fulfilled. This article shows that these assumptions can also be fulfilled in the case of path-oriented test data generation and presents a technique which can be used to generate test data covering certain paths in class methods.

1 Introduction

Testing is one of the vital activities of software development. It is conducted by executing the program developed with test inputs and comparing the observed output with the expected. As the input space of the program under test might be very large, testing has to be conducted with a representative subset. As an important consequence, testing cannot show the correctness of the program but rather shows that the program behaves as intended for the subset of the input space used.

Even if only a subset of the input space is used for testing, the effort required is still enormous. It is often claimed that testing and debugging accounts for approximately 50% of software development costs, which makes the need for automated testing support obvious.

This article presents a novel approach to automated test data generation. Several approaches have been proposed for automated test data generation, mainly *random*, *path-oriented*, *goal-oriented* and *intelligent* approaches [5]. Random techniques determine test data based on assumptions concerning fault distribution (e.g. [1]). Path-oriented techniques generally use control flow information to identify a set of paths to be covered and generate the appropriate test data for these paths. These techniques can further be classified in *static* and *dynamic* ones. Static techniques are often based on symbolic execution (e.g. [6]), whereas dynamic techniques obtain the necessary data by executing the program under test (e.g. [3, 4]). Goal-oriented techniques identify test data covering a selected goal such as a statement or a branch, irrespective of the path taken (e.g. [5]). Intelligent techniques of automated test data generation rely on complex computations to identify test data (e.g. [8]). Another classification of automated test data generation techniques can be found in [8].

The technique proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that of Korel [3]. The path to be covered is considered step-by-step, i.e. the goal of covering a path is divided into sub-goals, test data is then searched to fulfill them. The search process, however, differs substantially. Korel proposed searching test data minimizing a specific error function. In our approach, test data is determined using binary search, which requires certain assumptions but allows efficient test data generation.

The technique presented focuses on generating test data for class-level testing, i.e. it aims at identifying test data executing certain paths in class methods. As usually the path taken does not only depend on the method input but

*The chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

```
 1 public class SquareRoot {
 2   int maxiter;
 3
 4   SquareRoot(int maxiter) {
 5     this.maxiter = maxiter;
 6   }
 7
 8   double calc(double a, double eps) {
 9     double a1, a2, a3, a4, a5, b;
10     int iter;
11
12     if (a<=0.0 || eps<=0.0 || maxiter<=0) {
13       b = a + 3.0;              // for demonstration
14       if (Math.abs(b) == 3.0)   // purposes
15         a3 = 0.0;
16       else {
17         System.out.println("Wrong param. choice!");
18         a3 = -1.0;
19       }
20     }
21     else {
22       a1 = 0.0;
23       a2 = a;
24       iter = 0;
25       do {
26         a3 = a2-(a2*a2-a)*(a2-a1)/((a2*a2-a)-(a1*a1-a));
27         a1 = a3;
28         iter++;
29       } while (Math.abs(a2-a1)>=eps
30                && Math.abs(a3*a3-a)>=eps
31                && iter<maxiter);
32     }
33     return a3;
34   }
35 }
```

Figure 1. Source code of class SquareRoot

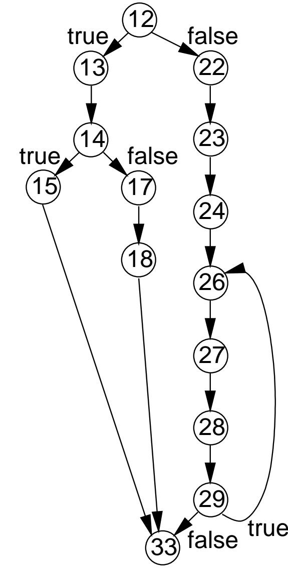

Figure 2. Flow graph of method calc()

also on the current object state, the generated test data also includes the necessary values of the instance variables.

This article consists of six sections. Section two introduces the terminology and explains the underlying concepts of the test data generation technique. Section three briefly describes the tool developed for automated test data generation. Section five demonstrates the tool on a sample class. Finally, section six presents our conclusion.

2 Terminology and Concepts

Let M be the method for which test data needs to be generated. The *control flow graph* of M is a directed graph (N, A, C, s, e) where N is a set of nodes, A is a subset of $N \times N$ referred to as a set of edges, C is a subset of $A \times \{true, false\}$ referred to as a set of branch conditions, and $s, e \in N$ are the unique entry and exit nodes, respectively. Nodes of a control flow graph correspond to statements, whereas edges correspond to control transfer among statements. More precisely, an edge $(n, n') \in A$ represents control transfer from statement n to n'. An edge (n, n') is called *branch* if n corresponds to the testing expression of an if- or loop statement. C includes for each branch (n, n') a $((n, n'), c)$ with c giving the target value of testing expression n necessary for traversal of the branch.

Figure 1 shows a class, called `SquareRoot`, implemented in Java. This class encapsulates an instance variable, called `maxiter`, and provides two methods, namely `SquareRoot()`, which represents its constructor, and `calc()`, which implements the Secant method for square root calculation in a given accuracy. The control flow graph of method `calc()` is given in figure 2. Nodes of the control flow graph are represented by circles and are annotated with numbers indicating the line number of the corresponding statement. Edges are shown as arrows. In case of an edges being a branch, it is augmented with the target value of the corresponding testing expression. For instance, branch $(12, 13)$ is augmented with `true` indicating the necessary target value of the testing expression at line 12.

The task of generating test data using dynamic path-oriented techniques mainly consists of three steps:

1. A control flow graph of the program under test is generated. In our case, the control flow graph required represents the method for which test data needs to be generated.

2. As the next step, a path is selected based on certain criteria, such as those based on data flow [7].

3. Finally, a search strategy is applied to identify test data covering the path selected.

In this article we focus on the third step. We particularly assume that the path to be covered has been determined be-

forehand based on some criteria. Infeasible paths should be avoided as far as possible, as in such a case a large number of iterations might be carried out without a result. Infeasible paths are a general problem in path-oriented test data generation [5].

Let O be the object providing method M and P the path in the control flow graph of M selected for traversal. P is defined as a vector $(n_1, \ldots, n_l)$ of statements with $n_1 = s$, $n_l = e$ and $(n_k, n_{k+1}) \in A$ for $1 \leq k < l$. Traversal of P usually does not only depend on the input of M but also on the current state of O, which is defined by its instance variables. For instance, path $(12, 13, 14, 17, 18, 33)$ is executed if at least one of the following conditions is fulfilled: input variable `a` is less than 0, input variable `eps` is less than or equal to 0, or instance variable `maxiter` is less than or equal to 0. It is therefore necessary to identify the appropriate values of instance variables besides identifying those of method input parameters. Let test data $x = (x_1, \ldots, x_n)$ be a vector of method input and object instance variables $x_1, \ldots, x_n$. The domain D_{x_i} of variable x_i is the set of all values x_i can hold. The input domain D of M is defined as the cross product $D_{x_1} \times \ldots \times D_{x_n}$. The problem we consider in this article is the identification of test data $x \in D$ executing P. A sufficient condition for the execution of P is that the testing expression of each if- and loop statements on P equals its necessary target value. Test data generation can thus be conducted by successively considering the corresponding branches and their testing expressions, and identifying test data ensuring the target values of the testing expressions. We focus therefore our consideration to a branch (n_k, n_{k+1}) on P and explain how x can be set so that testing expression n_k results its target value and the branch is traversed.

The important assumption of our approach is that there is, at least piece-wise, a monotone relation between test data x and testing expression n_k. Having a monotone relation, binary search techniques can be employed to determine a value for which the testing expression results the intended target value.

Let $n_k(x)$ be the value of the testing expression given test data x. Furthermore, let $\leq_{D_{x_i}}$ be an order relation defined on D_{x_i} and $\leq$ be an order relation on set $\{true, false\}$. Testing expression n_k is monotone on interval $I \subseteq D_{x_i}$ if for all $x_i, x_j \in I, x_i \leq_{D_{x_i}} x_j$ implies either

$$n_k((x_1, \ldots, x_i, \ldots, x_n)) \leq n_k((x_1, \ldots, x_j, \ldots, x_n))$$

$$\text{or } n_k((x_1, \ldots, x_j, \ldots, x_n)) \leq n_k((x_1, \ldots, x_i, \ldots, x_n)).$$

Monotony of a testing expression is defined with respect to the single variables x_i. As a testing expression might not be monotone on the entire domain D_{x_i} of a variable x_i but on single intervals, the monotony definition is restricted to intervals I of domain D_{x_i}. As a consequence, binary search has to be carried out for each test data variable separately and is restricted to these intervals if monotony cannot be ensured for the entire set.

Example. We consider as an example branch $(12, 13)$ in figure 2. The corresponding testing expression is defined as

```
a<=0.0 || eps<=0.0 || maxiter<=0
```

The test data generation algorithm starts by considering test data variable `a` and keeping the others constant. As the order relation '`<=`' is monotone on the entire domain of `a`, binary search does not need to be restricted to intervals of `a`'s domain. We assume, however, that the tester initializes the search interval to $[-10, 20]$, and sets `eps` $= 0.001$ and `maxiter` $= 100$. The binary search first determines the middle element of the initial interval, which is 5, and executes M with $x = (5, 0.001, 100)$. This value obviously does not traverse branch $(12, 13)$, as the testing expression at line 12 does not result the necessary target value. It then divides the search interval in two sub-intervals and continues the search with one of them according to the binary search approach. As the testing expression is monotone, the sub-interval to be continued with can be determining uniquely. In this case, the search is continued with interval $[-10, 5]$. Again, it determines its middle element, which is -2.5, and executes M with $x = (-2.5, 0.001, 100)$. In this case, the testing expression at line 12 results `true`, i.e. the target value, and the branch is traversed.

In the above example, the testing expression was monotone on the entire domain of `a` and it was thus not necessary to restrict binary search to intervals of its domain. In more general cases, however, this might not be the case and it might be necessary to consider intervals. A problem in such a case is the identification of interval boundaries. For instance, the testing expression at line 14 in figure 1 is defined as '`Math.abs(b) == 3.0`'. Obviously, '`Math.abs()`' is not monotone on its entire input domain but on intervals $(-\infty, 0]$ and $(0, \infty)$. In this case, we do not know how `b` has been obtained and which method inputs correspond to intervals boundaries $-\infty$, 0 and ∞. This problem is tackled in our approach as follows:

1. The input domain of each statement is partitioned into intervals on which the statement is monotone. For each statement, an interval is chosen.

2. The method under test is augmented with tracing statements to track the input of each statement.

3. During binary search, the input of each statement is observed and is checked whether it lies within the interval chosen before.

4. In the case of a statement input which does not lie in the corresponding interval, binary search is continued with test data increased or decreased accordingly.

5. In the case of binary search cannot identify appropriate test data, the steps above a repeated using a new set of chosen intervals.

This approach utilizes the fact that the composition of monotone functions yields again a monotone function. As in our case the functions, i.e. the statements, might only be monotone on certain intervals in their input domains, we need to ensure that the input of each statement lies within a particular interval.

Example. Consider path $(12, 13, 14, 15, 33)$. We assume that binary search is employed with respect to input variable `a` and $(-2.5, 0.001, 100)$ has been obtained traversing branch $(12, 13)$. Next, binary search is continued with respect to the same variable to identify an input traversing branch $(14, 15)$. The testing expression at line 14 is monotone on $(-\infty, 0]$ and $(0, \infty)$, and we assume that the first interval has been chosen according to step 1. During the execution of the method with input $(-2.5, 0.001, 100)$, it can observed that the testing expression at line 14 is evaluated on 0.5. Obviously, 0.5 does not lie within interval $(-\infty, 0]$, i.e. input variable `a` has to be decreased. As the tester has initialized the search interval to $[-10, 20]$, binary search is continued on interval $[-10, -2.5]$.

3 Test Data Generation Tool

A tool has been developed implementing the approach explained in the last section. This tool has been developed in Java and consists of five classes, namely `ClassUnderTest`, `Interval`, `Factory`, `Specifications` and `Generator`.

`ClassUnderTest` is an abstract class which has to be extended by the class under consideration. `ClassUnderTest` contains an abstract method called `methodUnderTest()` which needs to be overwritten by the method to be tested. Furthermore, this class also provides three methods called `execute()`, `statement()` and `testingExpression()`. Method `execute()` is invoked by methods of class `Generator()` for executing the method with a certain input and in a certain object state. It carries out initialization tasks necessary for tracing method execution and invokes `methodUnderTest()`. Method `statement()` is used to trace the execution of the method. It is inserted before each statement except the testing expression of if- and loop statements and is invoked with data necessary for tracing the statement. Method `testingExpression()` is inserted before testing expressions and is invoked with their input. This method is required to determine whether the test data considered ensured the target value of a testing expression.

`Interval` represents an interval in the input domain of a method or a statement. It provides methods for dividing an interval into two sub-intervals (`lowerInterval()` and `upperInterval()`). Furthermore, it also contains a method to determine the middle element of an interval for the purpose of the binary search algorithm (`midValue()`).

Class `Factory` provides a method called `midValue()`, which is required by methods of class `Interval`. It determines for two given objects of a class a third one which lies between them. The class needs two implement the `Comparable` interface to allow comparison of objects. Class `Factory` includes handling code for standard classes such as `Double`, `Integer` and `String`. Handling code for new classes, however, has to be added by the tester as part of the input for the automated test data generation.

`Specifications` provides methods for storing (`put()`) and retrieving (`get()`) the intervals of a statement on which it is monotone. Similar to class `Factory`, `Specifications` already contains the corresponding intervals of standard statements, i.e. methods of standard Java classes. The corresponding methods of newly developed classes, however, have to be added by the tester.

Class `Generator` implements as the main class the binary search. As input for the automated test data generation, the tester needs to specify the target values of the testing expressions on the path to be covered and needs to provide a starting interval for each variable. This input is passed together with an integer value defining the set of intervals to be considered to method `generateTestData`. This method conducts the binary search. It determines the middle elements of the initial input intervals and executes the method to be tested using the `execute()` method of class `ClassUnderTest`. Based on a flag returned by the `execute()` method, the interval of the variable considered is divided into two sub-intervals and the search is continued in one of them. The flag returned by method `execute()` indicates the direction in which the search variable has to be changed.

4 Case Study

As a case study we consider in the following class `SquareRoot` given in figure 1. As a first step, the tester needs to identify the path which is to be covered by test data. This can be done using the control flow graph of the method, which is given in figure 2. We assume that the tester has decided based on certain test criteria to generate test data for path $(12, 22, 23, 24, 26, 27, 28, 29, 26, 27, 28, 29, 33)$. The testing expression at line 12 is executed once and that at line 29 twice. The necessary target values of these testing expression are thus $(false, true, false)$.

As the next step, each statement on that path is decomposed, if possible, to atomic ones. Figure 3 show class `SquareRoot` after the decomposition step. The decom-

```
double calc(double a, double eps) {
  double a1, a2, a3, a4, a5;
  int iter;

  B0 = a<=0.0;
  B1 = eps<=0.0;
  B2 = maxiter<=0;
  B3 = B0 || B1;
  B4 = B3 || B2;
  if (B4) {
    B10 = a == 0.0;
    if (B10)
      a3 = 0.0;
    else {
      System.out.println("Wrong param. choice!");
      a3 = -1.0;
    }
  }
  else {
    a1 = 0.0;
    a2 = a;
    iter = 0;
    do {
      A0 = a2*a2;
      A1 = A0-a;
      A2 = a1*a1;
      A3 = a2-a1;
      A4 = A2-a;
      A5 = A1-A4;
      A6 = A1*A3;
      A7 = A6/A5;
      a3 = a2-A7;
      a1 = a3;
      iter++;
      A7 = a2-a1;
      A8 = Math.abs(A7);
      B5 = A8>=eps;
      A9 = a3*a3;
      A10 = A9-a;
      A11 = Math.abs(A10);
      B6 = A11>=eps;
      B7 = iter<maxiter;
      B8 = B5 && B6;
      B9 = B8 && B7;
    } while (B9);
  }
  return a3;
}
```

Figure 3. Class SquareRoot after decomposition of statements

position step can be carried out automatically using syntax analysis techniques. The tool in the current version, however, does not support this step yet. Decomposition of complex statements is necessary as the `Specifications` object only contains the necessary data concerning the intervals solely for atomic statements.

However, for some statements, which cannot be further decomposed, this data might nevertheless not be available. Particularly for a statement representing a newly developed method, the data concerning the intervals on which it is monotone has to be added to the `Specificaions` objects by the tester.

Method `calc()` is then augmented with tracing code to track the input of each statement and code necessary to extend `ClassUnderTest`. Even though the current version of the prototype does not automate this, this step can also be conducted automatically. For instance, tracing statement

```
statement("<", "B7", "iter", new Integer(iter),
   "maxiter", new Integer(maxiter))
```

is inserted before statement

```
B7 = iter<maxiter
```

to specify the statement, the variables and their values, and

```
testingExpression("B4")
```

is inserted before testing expression

```
if (B4)
```

to compare its current and target values.

Furthermore, it can also be necessary to extend the `Factory` class. The `midValue()` method of this class already contains handling code for Java standard classes. The input of method `calc()` consists of two `double` values and the object state. As necessary handling code for `Double` objects, corresponding to `double` values, are already available, the tester does not need to consider them. Method `midValue()`, however, does not include handling code for computing the object being in the middle of two `SquareRoot` objects. In case of this class, this can be carried out by creating an object with a `maxiter` instance variable having a value equal to the mean of the other objects' instance variables.

After these preparation steps, the corresponding method of the `Generator` class can be invoked and the test data generation can be started. This method, as mentioned before, requires as input the initial interval of each input variable. In our case, we assume that the tester uses interval $[-10.0, 100.0]$ for both `a` and `eps`, and interval $[0, 10000]$ for `maxiter`. In this case, an appropriate method input can be found after six executions of method `calc()`. The output of the test data generation algorithm is:

```
[-10.0, 100.0]  [-10.0, 100.0] [0, 10000]
[45.0, 100.0]   [-10.0, 100.0] [0, 10000]
[45.0, 72.5]    [-10.0, 100.0] [0, 10000]
[45.0, 58.75]   [-10.0, 100.0] [0, 10000]
[45.0, 51.875]  [-10.0, 100.0] [0, 10000]
[45.0, 48.4375] [-10.0, 100.0] [0, 10000]
Iterations: 6
Variable 0: 46.71875
Variable 1: 45.0
Variable 2: 5000
```

5 Conclusions

We have presented in this article a novel approach for test data generation based on binary search. Furthermore, a tool has been demonstrated implementing the approach for test data generation in the case of class-level testing.

We are continuing our research on this approach, as it possesses several benefits:

1. Although we have demonstrated the approach for generating numerical test data, it can be used to generate test data of any class. The only requirement is that the objects of the corresponding class are comparable with each other and a method is provided to obtain the object lying between two others.

2. It can be used for class-level testing. Object-oriented programming languages are used more and more in recent years for software development, making appropriate testing techniques necessary.

3. The success of some existing test data generation techniques often depends on certain parameters, i.e. we can encounter the problem of calibration. Our approach does not require parameter calibration.

4. Path-oriented test data generation is often carried out using optimizing techniques. Optimizing techniques can suffer from the problem of local minima or the initial starting point being too far from the solution [2]. Our approach does not suffer from these problems.

However, a problem can occur in the case of a method including several statements whose input domains have to be divided into intervals. As each combination of intervals has to be considered in the worst case, a large number of binary searches might be carried out before the appropriate input is identified.

Besides research on the concepts, we are also further developing the tool. There are a number of open issues which have to be addressed as mentioned in the case study.

Although the approach seems to be efficient, empirical studies are required to underline this claim. Additionally, we also need to compare the approach to existing test data generation techniques as far as possible. Another task in the future is therefore to conduct empirical studies.

References

[1] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, 1995.

[2] M. J. Gallagher and V. L. Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

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

[4] N. Mansour, M. Salame, and R. Joumaa. Integer- and real-value test generation for path coverage using a genetic algorithm. In *Software Engineering and Applications Conference (SEA)*, pages 49–54, 2000.

[5] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

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

[7] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[8] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, volume 23 of *Software Engineering Notes*, pages 73–81. ACM Press, 1998.