BINTEST – Binary Search-based Test Case Generation*

Sami Beydeda, Volker Gruhn
University of Leipzig
Department of Computer Science
Chair of 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 test cases. Various approaches have been proposed to automate this task. The approaches available, however, often have problems limiting their use. A problem of dynamic test case generation approaches, for instance, is that a large number of iterations can be necessary to obtain test cases. This article proposes a novel algorithm for path-oriented test case generation based on binary search and describes a possible implementation.

1 Introduction

Various approaches have been proposed for automated test case generation. The approaches available to date are classified in [9] into the categories *random*, *path-oriented*, *goal-oriented* and *intelligent* approaches. Random techniques determine test cases based on assumptions concerning fault distribution (e.g. [2]). Path-oriented techniques generally use control flow information to identify a set of paths to be covered and generate the appropriate test cases. These techniques can further be classified in *static* and *dynamic* ones. Static techniques are often based on symbolic execution (e.g. [10]), whereas dynamic techniques obtain the necessary data by executing the program under test (e.g. [8]). Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken (e.g. [9]). Intelligent techniques of automated test case generation rely on complex computations to identify test cases (e.g. [11]). Another classification of automated test case generation techniques can be found in [11].

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

The algorithm proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that of described in [8]. The path to be covered is considered step-by-step, i.e. the goal of covering a path is divided into subgoals, test cases are then searched to fulfill them. The search process, however, differs substantially. In [8], the search process is conducted a specific error function. In our approach, test cases are determined using binary search, which requires certain assumptions but allows efficient test case generation. This article explains the basic formal foundation of the *binary search-based test case generation (BINTEST) algorithm* together with a brief overview of a possible implementation. A more thorough explanation of the BINTEST algorithm, its implementation and other aspects can be found in [3, 5, 6].

2 Binary search in test case generation

2.1 Terminology

The primary use of the BINTEST algorithm is in testing the methods of a class. Before explaining the BINTEST algorithm, we first formally define the basic terms.

Def. 1. Let M be the method under test and C be the class providing this method. Let further $x_1, \ldots, x_n$ designate the arguments of M and attributes of C, and D_{x_i} with $1 \leq i \leq n$ be the set of all values which can be assigned to x_i. The *domain* D of M is defined as the cross product $D_{x_1} \times \cdots \times D_{x_n}$ and an *input* of M as an element x in D. A *test case* x_O is an input which satisfies a testing-relevant objective O. □

The above definition of the test case term is not restricted to arguments of M, it also encompasses the attributes of the class. The behavior of a method might not only depend on values of the method's arguments, it might also depend on values of the class' attributes.

The BINTEST algorithm generates test cases with respect to certain paths in the control flow graph of the method to be tested. It attempts to identify for a path P a test case x_P in the method's domain which is appropriate for the traversal of that path. The traversal of path P thus represents the testing-relevant objective O referred to in Def. 1. A test case x_P meets the objective of traversing path P if P is traversed when the arguments of M and attributes of C are set to the values specified by x_P and M is invoked. The definition below formalizes the terms used in the following.

Def. 2. The *control flow graph* G of M is a directed graph (V, E, s, e) where V is a set of nodes, $E \subseteq V^2$ is a set of edges, $s, e \in V$ are the initial and final node, respectively. A *segment* S in G with length l is defined as a vector $(v_1, \ldots, v_l) \in V^l$ of nodes with $(v_k, v_{k+1}) \in E$ for $1 \leq k < l$. A *path* P in G is a segment $(v_1, \ldots, v_m)$ with v_1 and v_m being the initial node s and final node e of G, respectively. □

The nodes of a control flow graph represent basic blocks within the source code of M, the edges control flow between basic blocks. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end [1].

2.2 Test case generation strategy

The BINTEST algorithm employs a strategy similar to that of the test case generation approach proposed in [8]: the path to be covered is not considered as a whole, it is rather divided into its basic constituents, in our case its edges, which are then considered in the sequence respecting their ordering on the path. A test case x_P is approached to by successively generating inputs covering each of the edges of P. A particular edge is only traversed by a subset of all possible inputs. The inputs need to fulfill the branching condition of this edge for its traversal.

Def. 3. Let $\mathbb{B}$ be the set of boolean values $\{true, false\}$. The *branching condition* $Br_{(v_k, v_{k+1})}$ of edge (v_k, v_{k+1}) is a function $Br_{(v_k, v_{k+1})} : D \rightarrow \mathbb{B}$ defined as

$$Br_{(v_k, v_{k+1})}(x) = \begin{cases} true & \text{if } v_k \text{ has one successor,} \\ b_{v_k}(x) = (v_k, v_{k+1}) & \text{else} \end{cases}$$

with $b_{v_k} : D \rightarrow E$ being a function which maps input x to the edge which connects v_k and the node representing the basic block entered after the traversal of that represented by v_k. □

Branching conditions associated with edges link single inputs in the method's domain with paths in the control flow graph. Control flow graphs as introduced so far do not relate inputs of the represented method with paths and we thus cannot determine the path traversed for a certain input.

The above definition of a branching condition distinguishes the case in which basic block v_k possesses a single successor from that in which it possesses two or more alternative successors. In the latter case, the basic block modeled by v_k includes as the last statement a branching statement, such as an `if`-, `for`- or `while`-statement in Java which necessitate two edges originating from a node and a `switch`-statement which can also necessitates more than two edges. A node, however, can also have no successors at all when control flow halts at the last statement of the basic block modeled, i.e. the node corresponds to the final node of the control flow graph. This case is excluded from consideration, since edges cannot originate from the final node.

Let $\mathcal{P}$ be the set of paths for which test cases need to be generated. The BINTEST technique receives the initial input $x^{(0)}$ from the tester and evaluates the branching condition of the first edge (v_1, v_2) of P, a path in $\mathcal{P}$, with respect to this value. Branching condition $Br_{(v_1, v_2)}$ is generally not met for all inputs in D but for values in a certain subset $D^{(1)} \subseteq D$ and the initial input is therefore changed to a value $x^{(1)} \in D^{(1)}$ ensuring the traversal of edge (v_1, v_2), i.e. $Br_{(v_1, v_2)}(x^{(1)}) = true$. In the next step, the branching condition of the second edge (v_2, v_3) on P is evaluated given that arguments of M and attributes of C are set to the values specified by $x^{(1)}$. Again, $Br_{(v_2, v_3)}$ is generally only satisfied by a subset $D^{(2)} \subseteq D$ and $x^{(1)}$ needs to be modified if it does not lie in $D^{(2)}$. $D^{(2)}$ is also a subset of $D^{(1)}$, since all values in $D^{(2)}$ have also to satisfy the branching condition of the predecessor edge. Hence, $D^{(2)} \subseteq D^{(1)} \subseteq D$. This procedure is continued for all edges on P until either an input is found fulfilling all branching conditions or a contradiction among these conditions is detected. In such a case, the branching conditions cannot be fulfilled entirely and the path is infeasible [11].

2.3 Monotony

The BINTEST algorithm approaches to a test case x_P traversing a path $P \in \mathcal{P}$ by successively generating a series of inputs $x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x_P$. In such a series, input $x^{(0)}$ is provided by the tester as the initial starting value, the others are calculated by the algorithm. Let Δ be the necessary qualitative modification in order to obtain $x^{(j+1)}$ from $x^{(j)}$. A possibility to determine Δ is using information concerning the monotony behavior of the branching condition under consideration.

Def. 4. Let $\leq_X$ and $\leq_Y$ be order relations defined on sets X and Y, respectively, and x, x' be two arbitrary elements in a subset I of X with $x \leq_X x'$. A function $f : X \rightarrow Y$ is *monotone increasing on* I iff

$$f(x) \leq_Y f(x')$$

and *monotone decreasing on I* iff

$$f(x) \geq_Y f(x').$$

Function f is called *monotone on* I iff it is either monotone increasing or decreasing on I, and is called *piecewise monotone* iff X is entirely partitioned and f is monotone on each subset. □

The notion of monotony describes the behavior of a function in relation to a change of the input. It gives a qualitative indication whether outputs of the function move in the same direction as inputs or in the reverse direction. Considering a branching condition as a function whose monotony behavior is known, the direction in which the input needs to be moved to satisfy the branching condition can be determined uniquely. Eventually, a branching condition might not be monotone on its entire domain. In such a case, consideration has to be restricted to a subset on which it is monotone with the consequence that Δ can only be uniquely determined within this subset and might be different in others.

The definition of branching condition $Br_{(v_k,v_{k+1})}$ in Def. 3 distinguishes two cases. It distinguishes the case in which node v_k has a single successor from that in which it has several alternative successors. In the former case, the branching condition constantly returns $true$. The monotony behavior is obvious. In the latter case, the result of the branching condition is determined by comparison of function b_{v_k} with the expected edge. The monotony behavior of the branching condition is here not that obvious, particularly if b_{v_k} is composed of other functions. The following lemma gives a property of functions composed of piecewise monotone functions.

Lem. 1. Assume that $f_1 : X_1 \to Y_1, \ldots, f_n : X_n \to Y_n$ is a family of piecewise monotone functions with $Y_i \subseteq X_{i+1}$. Let $F_n : X_1 \to Y_n$ be a function defined as the composition $F_n = f_n \circ \cdots \circ f_1$. Under this assumption, F_n is also piecewise monotone. □

Proof. The lemma is shown by induction over n. Case $F_1 = f_1$. Function f_1 is piecewise monotone by assumption and F_1 has the same property, since it equals to f_1. Case $F_{i+1} = f_{i+1} \circ F_i$. The composed function F_i is piecewise monotone due to the assumption of induction and let I be a subset of its domain's partition. Let x and x' be two arbitrary elements in I with $x \leq_{X_i} x'$. The monotony implies that one of the monotony conditions holds, i.e. either

$$F_i(x) \leq_{Y_i} F_i(x')$$

$$\text{or } F_i(x) \geq_{Y_i} F_i(x').$$

For the sake of brevity, this is also abbreviated as

$$F_i(x) \lessgtr_{Y_i} F_i(x').$$

Function f_{i+1} is also piecewise monotone by assumption. The monotony condition is satisfied by $F_i(x)$ and $F_i(x')$ if both lie in the same subset I' of its domain's partition. In such a case, following holds

$$\begin{aligned} f_{i+1}(F_i(x)) &\lessgtr_{X_{i+1}} f_{i+1}(F_i(x')) \\ f_{i+1} \circ F_i(x) &\lessgtr_{X_{i+1}} f_{i+1} \circ F_i(x') \\ F_{i+1}(x) &\lessgtr_{X_{i+1}} F_{i+1}(x') \end{aligned}$$

and F_{i+1} is also monotone on I. $F_i(x)$ and $F_i(x')$, however, might not be in I' and might thus not meet one of the monotony conditions. In such a case, F_{i+1} is not monotone on I and I needs to be partitioned in at most three subsets. One of these subsets contains the elements mapped on I', one of them the elements being less than the least and one of them the elements being greater than the greatest element of the first subset. F_{i+1} is then monotone on all three subsets and is therefore piecewise monotone. □

A function f_i is referred to as *atomic* if it cannot be further decomposed. Such a function is typically implemented by the underlying programming language as an operation or by a class or component as a method. The application of the lemma to Def. 3 is obvious. The comparison of b_{v_k} with the expected edge can be considered as the composed function $F = f_n \circ \cdots \circ f_1$ with f_n being the comparison and $f_{n-1} \circ \cdots \circ f_1$ being the decomposed form of b_{v_k}, if it is not already atomic. The single atomic functions of which a particular branching condition is composed can technically be determined using tracing. Tracing statements can be inserted into the method under test which are executed immediately prior to the execution of operations and methods corresponding to atomic functions, and the atomic functions constituting a particular branching condition can be identified.

2.4 BINTEST algorithm

The necessary modification to obtain a test case x_P is indicated by Δ qualitatively. Δ gives the direction in which the input needs to be modified, not the extend of the modification and x_P thus cannot be directly obtained. x_P needs therefore to be approached to iteratively. The BINTEST algorithm conducts the necessary iterations according to the *binary search strategy*.

Fig. 1 shows the BINTEST algorithm. The BINTEST algorithm requires as input the set of paths $\mathcal{P}$ to be traversed and computes an appropriate set of test cases T as output. Line 1 of the algorithm has the purpose of initialization, in this line the set T is initialized in which the test cases covering the paths in $\mathcal{P}$ are stored. After initialization, lines 3–41 are executed for each path P to be covered. For each path P, a set A is constructed including all possible vectors $(I_1, \ldots, I_{m-1})$ with I_i being a subset in the partition of the branching condition's domain associated with the ith edge (line 3). After the definition of A, a repeat-loop is entered

1: $T = \emptyset$
2: for each path $P = (v_1, \ldots, v_m) \in \mathcal{P}$
3: $A = A_1 \times \cdots \times A_{m-1}$, with A_i being the partition of $Br_{(v_i, v_{i+1})}$'s domain
4: repeat
5: select a $(I_1, \ldots, I_{m-1})$ from A
6: remove $(I_1, \ldots, I_{m-1})$ from A
7: $I = D$
8: $monotony = increasing$
9: $k' = 0$
10: repeat
11: determine middle element x of I
12: consider the edges on P according to their ordering and determine the first edge (v_k, v_{k+1})
with either $x \notin I_k$ or $Br_{(v_k, v_{k+1})}(x) \neq true$ if such an edge exists
13: if edge (v_k, v_{k+1}) exists
14: then
15: if $k = k'$ and ($\Delta' = increase$ and $x < x'$ or $\Delta' = decrease$ and $x > x'$)
16: then
17: $I = I_{other}$
18: if $monotony = increasing$
19: then $monotony = decreasing$
20: else $monotony = increasing$
21: else
22: if $x <$ lower boundary of I_k or $Br_{(v_k, v_{k+1})}(x) \neq true$
23: then $\Delta = increase$
24: else $\Delta = decrease$
25: bisect I in two halves I_{low} and I_{up}
26: if $\Delta = increase$ and $monotony = increasing$ or $\Delta = decrease$ and $monotony = decreasing$
27: then
28: $I = I_{up}$
29: $I_{other} = I_{low}$
30: else
31: $I = I_{low}$
33: $I_{other} = I_{up}$
34: $k' = k$
35: $\Delta' = \Delta$
37: $x' = x$
38: until edge (v_k, v_{k+1}) does not exist or $I = \emptyset$
39: until edge (v_k, v_{k+1}) does not exist or $A = \emptyset$
40: if edge (v_k, v_{k+1}) does not exist
41: then add x to T
42: return T

Figure 1. The BINTEST algorithm.

in which the elements in A are considered until either a test case has been found covering P or A is empty and thus P is infeasible (lines 4–39). The following lines of the algorithm, lines 5–9, initialize the variables used in an iteration. A vector is selected and remove from A, the subset I in which the test case is searched is set to the method's domain D and the branching condition considered is assumed to be monotone increasing. After this initialization, lines 10–38 are executed for the vector selected until an appropriate test case is found or the subset in which the search is conducted has been bisected to the empty set and thus a test case could not be found for the vector considered. Within this loop, the middle element x of the search subset is computed (line 11). The branching conditions on P are then considered regarding to x starting with that of the first edge on P and proceeding with respect to the ordering of the edges on P (line 12). Each branching condition is analyzed to determine whether x lies in the subset specified by the vector and the branching condition results $true$ with that input. If for a branching condition these conditions are not fulfilled, P is not covered and lines 14–36 are executed in order to fulfill them. The operations conducted in these lines aim at approaching

to an appropriate input. The algorithm validates in line 15 whether the monotony assumption is correct. For this purpose, it firstly determines whether the branching condition considered has also been considered in the last iteration by comparing their indexes k and k', and secondly determines whether the input x has moved in the direction suggested by Δ', the qualitative modification indication obtained in the last iteration. If the monotony assumption was wrong, the bisection step of the last iteration is reverted and the monotony assumption is corrected (lines 17–20). Lines 22–32 are executed if the monotony was correct. Here, a Δ is obtained by assessing if x lies in the subset prescribed by the vector for the branching condition considered and the branching condition is fulfilled (lines 22–32). After obtaining Δ, the search subset I is bisected and one of the halves is selected taking into account Δ and the monotony assumption (lines 26–33). Lines 34–37 of the algorithm in the remainder of the loop store the current state of the algorithm in auxiliary variables to allow the reversion of the bisection step if the monotony assumption appears in the next iteration to be wrong. After the execution of the loop, x is added to T if P has been covered.

3 Tool support

The BINTEST algorithm has been implemented as a part of a framework, *BINTEST framework*, supporting the testing of classes and components. The integration of the class under test with this framework encompasses two tasks, which are the following:

> Firstly, the framework requires that the class under consideration needs to implement a certain interface, called `ClassUnderTest`.
>
> Secondly, the methods which the user might wish to test need to be instrumented with tracing statements.

The two tasks cannot be fully automated. The first task in particular requires application context-specific information to implement the methods declared by the interface, which cannot be automatically deduced from the source code of the class. The methods to be implemented by the class consist of, for instance, those to determine the middle element between two others in the domain of the method under test, and to obtain the domains' partitions of operations and methods invoked during the execution of the method under test. Obviously, these methods cannot be automatically generated and need to be implemented by the tester. However, the tester does not need to implement these methods for standard operations of the Java programming languages and methods of the standard class library, since they are available by the framework.

Contrary to the first task, the second task which has to be performed during the preprocessing phase can be fully automated. This task consists of inserting tracing code for obtaining information for following purposes:

> Firstly, an objective of the tracing code is to deliver the information necessary to validate that inputs of the methods invoked during the execution of the method under test lie in the corresponding subsets of their domains. The information required consists of an ID identifying the method invoked, an ID identifying the invocation and the input passed.
>
> Secondly, the objective of the tracing code is also to collect the necessary information to determine the path traversed during execution of the method under test. The path traversed can be uniquely determined by observing the edges selected by the conditional statements. The edge selected by a conditional statement generally depends on a boolean expression and edge selected can identified by observing the values of these expressions.

The framework, or more clearly the main class of the framework, offers for this purpose the methods `I`, `E` and `T`. These methods have to be inserted at the appropriate positions in the source code of the method under test.

> `I`. This method has the purpose of observing the value of an arbitrary expression without interfering its execution. Such an expression might be an argument in a method call or the expression within a conditional statement according to which the alternative executed next is chosen. For instance, in the case of `f(x)` being a method invocation, the tracing statement has to be inserted in this method invocation so that the value of `x` is observed. It can be achieved by `f(I(x))`.
>
> `E`. This method has the purpose of identifying the ID of the method invoked again without interfering the invocation. This can be achieved by inserting an invocation of `E` right after the invocation of a method with explicitly giving its ID. In the case of the above example, this can be accomplished by the framework as `E("f", f(I(x)))`.
>
> `T`. This method can be considered as being the counterpart of `E` for the use in the context of conditional statements. Similar to method `E`, which indicates that a method has been invoked, method `T` indicates that a conditional statement has been reached. For instance, an `if`-statement such as `if (x==2) ...` this is accomplished by `if (T(1,(x==2))) ...` with `1` being the ID of this conditional statement.

The information necessary to conduct the second task of the preprocessing phase can generally be obtained by syntactical analysis of the source code of the method under test as also apparent by the above brief explanation of the three methods. This information can, for instance, be computed by tools capable to parse the Java programming language. Such a tool has been generated using the parser generator ANTLR 2.7.1. The tool generated parses the source code of the class and inserts the tracing statements at the appropriate positions.

4 Conclusions

We have presented in this article a novel approach for test case generation based on binary search and a tool implementing the approach for test data generation in the case of class-level testing. A case study has been conducted which has shown that the algorithm developed can require far less iterations than other comparable approaches.

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

1. We have demonstrated the approach for generating numerical test data. However, it can also be used to generate test cases of any class as long as the two basic assumptions are satisfied.

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 case generation techniques often depends on certain parameters and 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 [7]. 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.

Even if the approach is efficient in the case study considered, empirical studies are required to generalize this claim. Additionally, we also need to compare the approach to existing test case generation techniques as far as possible. Another task in the future are therefore comparative empirical studies.

One of the applications of the BINTEST algorithm is in testing COTS components [3, 4]. We believe that the diverging needs of the two parties involved in the development of a component-based system, component developers and developer of the system, can be met by self-testability. A possibility to achieve component self-testability is to embed a test case generation technique, such as the BINTEST approach, into the component. In this context, we would like to invite the reader to participate in an open discussion started to gain a consensus concerning the problems and open issues in testing components. The contributions received so far can be found at `http://www.stecc.de` and new contributions can be made by email to `sami.beydeda@informatik.uni-leipzig.de`.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison Wesley, 1988.

[2] 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.

[3] S. Beydeda. *The Self-Testing COTS Components (STECC) Method*. PhD thesis, Universität Leipzig, Fakultät für Mathematik und Informatik, 2003.

[4] S. Beydeda and V. Gruhn. Merging components and testing tools: The self-testing COTS components (STECC) strategy. In *EUROMICRO Conference Component-based Software Engineering Track*. IEEE Computer Society Press, 2003.

[5] S. Beydeda and V. Gruhn. Test case generation according to the binary search strategy. In *International Symposium on Computer and Information Sciences (ISCIS)*, LNCS. Springer Verlag, 2003.

[6] S. Beydeda and V. Gruhn. Test data generation based on binary search for class-level testing. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*. IEEE Computer Society Press, 2003.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.