Test Case Generation According to the Binary Search Strategy*

Sami Beydeda and 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
http://www.lpz-ebusiness.de

Abstract. One of the important tasks during software testing is the generation of test cases. Unfortunately, existing approaches to test case generation 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 introduces a formal framework for the application of the well-known search strategy of binary search in path-oriented test case generation and explains the *binary search-based test case generation (BINTEST) algorithm.*

1 Introduction

This article presents a novel approach to automated test case generation. Several approaches have been proposed for test case generation, mainly *random*, *path-oriented*, *goal-oriented* and *intelligent* approaches [8]. 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 for these paths. These techniques can further be classified in *static* and *dynamic* ones. Static techniques are often based on symbolic execution (e.g. [9]), whereas dynamic techniques obtain the necessary data by executing the program under test (e.g. [7]). Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken (e.g. [8]). Intelligent techniques of automated test case generation rely on complex computations to identify test cases (e.g. [10]). Another classification of automated test case generation techniques can be found in [10].

The algorithm proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that in [7]. 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

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

substantially. In [7], the search process is conducted according to a specific error function. In our approach, test cases are determined using binary search, which requires certain assumptions but allows efficient test case generation. Other aspects of the BINTEST approach are covered in [3, 4].

2 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.

Definition 1. *Let M be the method under test and C the class providing this method. Furthermore, let $a_1, \dots, a_l$ designate the arguments of M and attributes of C, and D_{a_i} with $1 \leq i \leq l$ be the set of all values which can be assigned to a_i.*

(i) *The* domain *D of M is defined as the cross product $D_{a_1} \times \cdots \times D_{a_l}$,*
(ii) *an* input *of M as an element x in D and*
(iii) *a* test case *x_O as an input which satisfies a testing-relevant objective O.*

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 appropriate for the traversal of that path. The traversal of path P thus represents the testing-relevant objective O referred to in Definition 1. A test case x_P is characterized as *traversing* or *covering* a path P if the blocks modeled by nodes constituting P are executed in the sequence prescribed if the arguments of M and attributes of C are set to the values specified by x_P, and M is invoked. The definition below defines some of the basic term required.

Definition 2. *Let M be the method under test.*

(i) *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, and $s, e \in V$ are the initial and final node, respectively. The initial and final node in a control flow graph are the only nodes having no predecessor and no successor, respectively.*
(iii) *A* path *P in G is defined as a tuple $(v_1, \dots, v_m) \in V^m$ of nodes with $(v_j, v_{j+1}) \in E$ for $1 \leq j < m$, v_1 and v_m being the initial node s and final node s 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].

3 Test case generation strategy

The BINTEST algorithm employs a strategy similar to that of the test case generation approach in [7]. In both approaches, the path to be covered is not considered as a whole, but rather divided into its basic constituents, which are then considered in the sequence respecting their order on the path. A test case x_P is approached by iteratively generating inputs successively 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 traversal condition of this edge.

Definition 3. *Let M be the method under test, D its domain and (v_j, v_{j+1}) an edge in its control flow graph. Let furthermore be $\mathbb{B}$ the set of boolean values $\{\mathit{false}, \mathit{true}\}$. The* traversal condition *$T_{(v_j,v_{j+1})}$ of edge (v_j, v_{j+1}) is a function $T_{(v_j,v_{j+1})} : D \to \mathbb{B}$ defined as*

$$T_{(v_j,v_{j+1})}(x) = \begin{cases} \mathit{true} \text{ if } (v_j, v_{j+1}) \text{ is traversed by } x, \\ \mathit{false} \text{ otherwise.} \end{cases}$$

Traversal conditions associated with edges of a control flow graph link single inputs in the method's domain with paths in its 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 BINTEST algorithm receives the initial input x_0 from the tester and evaluates the traversal condition of the first edge (v_1, v_2) of P with respect to this value. Traversal condition $T_{(v_1,v_2)}$ 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. $T_{(v_1,v_2)}(x_1) = \mathit{true}$. In the next step, the traversal 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, $T_{(v_2,v_3)}$ is generally only satisfied by a subset $D_2 \subseteq D_1$ and x_1 needs to be modified if it does not lie in D_2. 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 traversal conditions or a contradiction among these conditions is detected. In such a case, the traversal conditions cannot be fulfilled entirely and the path is infeasible [10].

4 Monotony

The BINTEST algorithm approaches to a test case x_P traversing a path P by iteratively 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 test case generation algorithm. Let Δ_i be the necessary qualitative modification in order to obtain x_{i+1} from x_i. A possibility to determine Δ_i is using information concerning the monotony behavior of the traversal condition under consideration.

Definition 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*

(i) monotone increasing on I *iff $f(x) \leq_Y f(x')$,*
(ii) monotone decreasing on I *iff $f(x) \geq_Y f(x')$ and*
(iii) monotone on *I iff it is either monotone increasing or decreasing on I.*

Definition 5. *Let X be a set and 2^X its power set.*

(i) *A* partition *A of X is a subset of 2^X with $\bigcup_{a \in A} = X$ and $\forall a_1, a_2 \in A, a_1 \neq a_2 : a_1 \cap a_2 = \emptyset$. The single sets in A are also called* blocks.
(ii) *A function $f : X \rightarrow Y$ is called* piecewise monotone *iff it is monotone on all blocks in a partition A.*

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 traversal condition as a function whose monotony behavior is known, the direction in which the input needs to be moved to satisfy the traversal condition can be determined uniquely. Eventually, a traversal 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_i can only be uniquely determined within this subset and might be different in others. Depending on the size of these subsets, they can also be grouped together according to the monotony behavior of the corresponding traversal condition to avoid that test case identification degenerates to linear search. For instance, a traversal condition defined on the set of integer numbers which maps even values to $true$ and odd values to $false$ is only monotone on subsets containing two elements. Instead of considering these subsets separately, they can be grouped to a family such as $(\{i, i+1\})_{i \text{ is integer}}$ and, since the traversal condition possesses the same monotony behavior on all subsets in the family, the family can be considered as a whole.

However, the monotony behavior of arbitrary traversal conditions is usually not known and suitable means are required to determine it. One solution is to explicitly specify the monotony behavior of the traversal condition associated with each edge in the control flow graph of a method. This solution is obviously undesirable, since the control flow graph of a complex method can have a large number of edges and the effort for monotony behavior specification can be substantial. The solution employed in the context of the BINTEST algorithm is to deduce the monotony behavior of a traversal condition from those of the functions of which the traversal condition is composed. The following lemma gives the formal basis for this.

Lemma 1. *Assume that $(f_k : X_k \rightarrow Y_k)_{1 \leq k \leq n}$ is a family of functions, with f_k being piecewise monotone with respect to order relations $\leq_{X_k}$ and $\leq_{Y_k}$, and $Y_k \subseteq X_{k+1}$. Let $F_n : X_1 \rightarrow Y_n$ 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 with respect to order relation $\leq_{X_1}$ defined on its domain and $\leq_{Y_n}$ defined on its codomain.*

The proof of this lemma has been omitted due to space restrictions and can be found in [3]. A function 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 single atomic functions of which a particular traversal 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 traversal condition can be identified.

5 Algorithm

Figure 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, $\mathcal{T}$, as output. The algorithm executes three distinct phases during its execution. These phase are the initialization phase (line 1), the test case generation phase (lines 2–45) and the finalization phase (line 46). The essential phase among them is obviously the test case generation phase. The algorithm executes three nested loops during this phase, which are the following:

Outmost loop. The single paths in $\mathcal{P}$ are considered during respective iterations of the outmost loop (lines 2–45). Each path $P \in \mathcal{P}$ is considered in an iteration, during which set A is initialized (line 3), the second loop is entered to generate an appropriate test case (lines 4–43) and the test case, if any could be generated, is added to $\mathcal{T}$ (lines 44–45). For the path $P = (v_1, \ldots, v_m)$ considered, a set A is constructed including all possible tuples $(I_1, \ldots, I_{m-1})$ with I_j being a block in the partition of $T_{(v_j,v_{j+1})}$'s domain. Block I_j is a subset of the domain of traversal condition $T_{(v_j,v_{j+1})}$ on which it is monotone and in which thus an appropriate test case can be approached to. The traversal of a path requires that all traversal conditions need to be considered in order to identify an input x satisfying all, or formally $\bigwedge_{1 \leq j < m} T_{(v_j,v_{j+1})}(x)$, the sufficient condition for the path's traversal. A block needs therefore to be specified for the traversal condition associated to each edge on the path. An appropriate input x, if it exists, thus necessarily lies in all of these blocks, i.e. $x \in \bigcap_{1 \leq j < m} I_j$.

Middle loop. The single tuples in A are considered during respective iterations of the second loop (lines 4–43). This loop is executed until either a test case has been found covering P or all elements in A have been considered and P is thus infeasible. During an iteration of this loop, an element $(I_1, ..., I_{m-1})$ is selected from A which has not been considered in an iteration before, and D_{cur} and j_{prev} are initialized (lines 5–7). D_{cur} gives the current search interval, whereas j_{prev} gives the index of the traversal condition considered in the previous iteration. After these initialization steps, the third and innermost loop is entered (lines 8–42).

```
1: $\mathcal{T} = \emptyset$
2: for each path $P = (v_1, \ldots, v_m) \in \mathcal{P}$
3:    $A = A_1 \times \cdots \times A_{m-1}$, with $A_j$ the partition of $T_{(v_j,v_{j+1})}$'s domain, $1 \leq j \leq m-1$
4:    repeat
5:       $(I_1, \ldots, I_{m-1})$ = an arbitrary element in $A$ not considered before
6:       $D_{cur} = D$
7:       $j_{prev} = 0$
8:       repeat
9:          $x_{mid}$ = middle element of $D_{cur}$ regarding to $\leq_D$
10:         $j_{min}$ = least $j$ with $x_{mid} \notin I_j$ or $T_{(v_j,v_{j+1})}(x_{mid}) \neq true$, 0 if $j$ does not exist
11:         if $j_{min} \neq 0$
12:         then
13:            if $x_{mid} \notin I_{j_{min}}$
14:            then
15:               if $x_{mid} <_D$ lower boundary of $I_j$
16:               then $\Delta = increase$
17:               else $\Delta = decrease$
18:            else
19:               if $j_{min} > j_{prev}$
20:               then
21:                  $monotony = increasing$
22:                  $\Delta = increase$
23:                  $D_{backup} = D_{cur}$
24:               if $j_{min} = j_{prev}$
25:               then
26:                  if $monotony = increasing$
27:                  then $\Delta = increase$
28:                  else $\Delta = decrease$
29:               if $j_{min} < j_{prev}$
30:               then
31:                  if $monotony = increasing$
32:                  then $\Delta = decrease$
33:                  else $\Delta = increase$
34:            if $\Delta = increase$
35:            then $D_{cur} = \{x \in D_{cur} \mid x \not\leq_D x_{mid}\}$
36:            else $D_{cur} = \{x \in D_{cur} \mid x \leq_D x_{mid}\}$
37:            if $D_{cur} = \emptyset$ and $monotony = increasing$
38:            then
39:               $monotony = decreasing$
40:               $D_{cur} = D_{backup}$
41:         $j_{prev} = j_{min}$
42:      until $j_{min} = 0$ or $D_{cur} = \emptyset$
43:   until $j_{min} = 0$ or all elements in $A$ have been considered
44:   if $j_{min} = 0$
45:   then add $x$ to $\mathcal{T}$
46: return $\mathcal{T}$
```

Fig. 1. The BINTEST algorithm.

Innermost loop. The binary search strategy is implemented by the innermost loop, which is executed until either an appropriate test case has been found or the search interval is empty and thus cannot contain an appropriate test case (lines 8–42). The first operation of an iteration is computation of the search interval's middle element x_{mid}. The traversal conditions associated to edges on P are then considered regarding to x_{mid} starting with that of the first edge on P and proceeding with respect to their ordering on P. Each traversal condition is analyzed to determine whether x_{mid} lies in the block specified by the tuple and the traversal condition results *true* for that input. j_{min} gives either the index of the first traversal condition for which one of these conditions is not satisfied, or its has a value of 0 to indicate that both conditions are satisfied for all traversal conditions. In the case of $j_{min} \neq 0$, i.e. P is not covered, lines 13–41 are executed in order to approach to an appropriate input. Otherwise, a test case has been found covering P and the inner two loops are left. The algorithm validates if x_{mid} is in the block $I_{j_{min}}$ specified by $(I_1, ..., I_{m-1})$ and sets Δ accordingly if it does not (lines 15–17). If x_{mid} lies in the block specified, j_{min} necessarily references the first traversal condition which is not satisfied by x_{mid} and the algorithm attempts to approach to an appropriate input (lines 19–33). For this purpose, j_{min} is compared with j_{prev}, the value of j_{min} during the last iteration. j_{min} can be greater, equal to or less than j_{prev}. The first case is given when the traversal condition considered during the last iteration is now satisfied and the algorithm considers in the current iteration the traversal condition associated to one of the next edges on the path (lines 19–23). As the traversal condition is considered for the first time, its monotony behavior is not known and an assumption is made. Based on this assumption Δ is set accordingly and the current search interval is stored in an auxiliary variable to allow the correction of the monotony assumption later. The second case is given when the traversal condition considered during the last iteration is still not fulfilled (lines 24–28). Another iteration is thereby necessary and Δ is set according to the monotony assumption made before. Finally, the third case occurs if a traversal condition satisfied before is not satisfied anymore due to the modification of the input made during the last iteration (lines 29–33). In this case, Δ is set to the reverse of the value assigned to it during the last iteration in order to approach to a value satisfying this traversal condition again. After obtaining Δ, the current search interval is bisected according to x_{mid} and one of the halves is selected with respect to Δ (lines 34–36). The iterations made, however, do not necessarily have to result in a suitable test case. One reason for this can be a wrong monotony assumption. The algorithm handles wrong monotony assumptions by reseting the search interval to that when the monotony assumption has been made and correcting the assumption (lines 37–40). Finally, the index of the traversal condition considered is stored in an auxiliary variable.

Theorem 1. *Let $P = (v_1, \ldots, v_m)$ be the path to be covered. Assume that the domains of the the traversal conditions $T_{(v_j, v_{j+1})}$, $1 \leq j < m$, are each partitioned in at most n blocks. The number of iterations conducted by the BINTEST algorithm to identify a test case covering P is bounded by $O(mn \; log \; |D|)$.*

The proof of this theorem has been omitted due to space restrictions and can be found in [3].

6 Conclusions

We have presented in this article a novel approach for test case generation based on binary search. We are continuing our research on this approach, as it possesses several benefits. One of these benefits is that it can be used to generate test cases of any type as long as a total order exists on the search interval. Furthermore, 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. Another aspect is that 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 [6]. Our approach does not suffer from these problems.

One of the applications of the BINTEST algorithm is in testing COTS components [3, 5]. 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. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison Wesley, 1988.
2. Alberto Avritzer and Elaine 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. Sami Beydeda. *The Self-Testing COTS Components (STECC) Method*. PhD thesis, Universität Leipzig, Fakultät für Mathematik und Informatik, 2003.
4. Sami Beydeda and Volker Gruhn. BINTEST – binary search-based test case generation. In *Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society Press, 2003.
5. Sami Beydeda and Volker 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.
6. Matthew J. Gallagher and V. Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.
7. Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
8. Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.
9. 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.
10. Nigel Tracey, John Clark, and Keith 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.