An Efficient Method for Automated Control Flow Testing of Programs

Quang-Trung Nguyen(✉) and Pham Ngoc-Hung

Faculty of Information Technology, VNU University of Engineering and Technology, Lahore, Pakistan
{trungnq.mi11,hungpn}@vnu.edu.vn

Abstract. This paper presents a method for automated control flow testing of unit programs. The key idea of this method is to combine the black-box and white-box techniques in order to minimize the complexity of white-box testing. Instead of performing black-box and white-box separately, the proposed method uses the test inputs that are generated by black-box to reduce searching space of white-box testing. The method then continually eliminates arcs in remaining space to find non-duplicated test paths. Therefore, the proposed method is able to operate white-box testing with less effort than the current method.

1 Introduction

Unit testing has been recognized as a key phase in improving software quality in practice. Two techniques to operate unit testing are white-box and black-box. Generally, black-box technique is inexpensive as well as easy to create test cases and to test unpredictable behavior. However, it cannot detect internal errors. On the contrary, control flow testing which has been known as a major technique of white-box is more efficient for this problem. Unfortunately, this technique requires deep understanding and high-level skill to analyze source code. Because it is performed manually so it is very costly and inefficient. It is believed that automatic testing is a solution to perform these types of testing more effectively. Indeed, two techniques black-box and white-box have difference pros and cons, and both of them, by them self, cannot supersede each other. In software companies, that black-box and white-box are performed separately takes a lot of time and effort. From the above-mentioned, what we need to find out is how to create an efficient automatic tool combining black-box's advantages and white-box's ones to produce a small set of test input data.

For recent years, there are many researches putting effort to minimize the size of test inputs. For example, Gupta and Soffa [5] have studied the ways of gathering coverage requirements so that each group can be covered by a test case, which followed by guiding test case generation to produces a test case satisfies multiple coverage requirements. Based on basis path testing concept [1], Ahmed S. Ghiduk, O. Said and Sultan Aljahdali [6] introduced strategy using genetic algorithm for automatically generating basis test paths. Bertolino and Marre [2]

P.C. Vinh et al. (Eds.): ICCASA 2013, LNICST 128, pp. 133–143, 2014.
DOI: 10.1007/978-3-319-05939-6_14,

proposed a path generation method by using a reduced CFG. Guangmei et al [3] presented an automatic generation method of basis set of paths which is built by searching the CFG by depth-first searching method. On the other hand, with spanning sets, Martina Marre and Antonia Bertolino [4] reduced and estimated the number of test cases to satisfy coverage criteria.

This paper proposes a method to take advantage of both black-box and while-box techniques with the purpose of performing control follow testing of programs efficiently. Instead of focusing on whole CFG of the given unit, this method only uses a simplified CFG of it. The simplified CFG has been generated by removing the testing paths that are covered by the test cases of the black-box testing. Hence, the size of CFG is significantly reduced, which is followed by reducing effort in white-box testing. In addition, this method avoids duplicated paths with graph reduced before and during the searching process. It is also potentially performed to reuse test cases in the context of program evolution. Consequently, software companies have method for getting high quality set of test inputs with low cost.

The paper is organized as follows. Firstly, we introduce definitions and theorems for reducing graph in Sect. 2. The method, and two major steps of the proposed method will be shown in Sects. 3 and 4. In Sect. 5 we evaluate time complexity of two algorithms mentioned in Sects. 3, 4, and the whole of our method. Finally, Sect. 6 is conclusion.

2 Background

In this section, we show some basic definitions and theorems of graph theory that will be used through the paper.

Control flow graph which is a directed graph represents graphically the information needed to select the test cases. A control flow graph $G = (N, A)$ consists of a set N of nodes or vertices, and a set A of directed edges or arcs, where a directed edge $e = (T(e), H(e))$ is an ordered pair of adjacent nodes, called Tail and Head of e, respectively: we say that e leaves $H(e)$ and enters $T(e)$. If $H(e) = T(e')$, e and e' are called adjacent arcs. For a node n in N, $indegree(n)$ is the number of arcs entering and $outdegree(n)$ the number of arcs leaving it.

A program control flow may be mapped onto a flow graph in different ways. In this paper, we use a flow graph representation called ddgraph (decision-to-decision graph) which is particularly suitable for the purposes of branch. The following is a formal definition of ddgraph.

Definition 1. *(Ddgraph). A ddgraph is a graph* $G = (N, A)$ *with two distinguished nodes* n_1 *and* n_0*(the unique entry node and the unique exit node, respectively), such that any node* $n \in N$ *is reached by* n_1 *and reaches* n_0*, and for each node* $n \in N$*, except* n_1 *and* n_0*,* $(indegree(n) + outdegree(n)) > 2$*, while* $indegree(n_1) = 0$*,* $outdegree(n_1) = 1$*,* $indegree(n_0) > 0$*,* $outdegree(n_0) = 0$*.*

A path P of length q in a ddgraph G is a sequence P=$(e_1, e_2, .., e_q)$ where $T(e_{i+1}) = H(e)$ for i = 1,. . . , q-1. P is said to be a path from e_1 to e_q, or

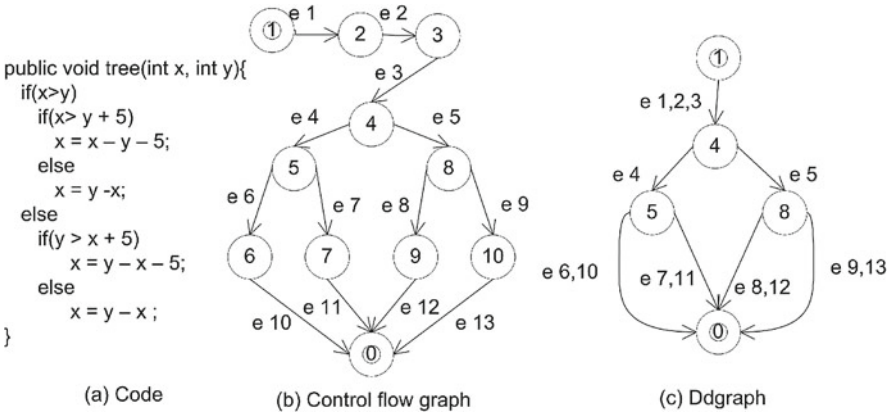

Fig. 1. Source code, control flow graph and ddgraph of tree method.

from H(e_1) to T(e_q). A node n reaches an arc n' if there is a path P in G from n to n'. A path, in ddgraph, is call complete path if it is path from n_1 to n_0.

Figure 1 shows the source code, traditional CFG and ddgraph form of a java unit. This method is the most complete case of ddgraph. In this case, the normal white-box technique needs select at least $|N| - 1$ path where $|N|$ is number of nodes in ddgraph for covering all branches of the unit.

Definition 2. *(Redundant arc). In ddgraph G, An arc e is call redundant arc if* $indegree(T(e) > 1$, $outdegree(H(e)) > 1$.

Theorem 1. *We can remove redundant arc without remove connective property of G.*

Proof. Assuming that we have a ddgraph $G = (N, A)$ with two distinguished nodes n_1 and n_0 and $e \in A$. By Definition 2, the arc e is redundant arc which $indegree(T(e)) > 1$. So, it is clear that, when we eliminate arc e in G, we can find at least one path from n_1 to T(e) , it mean that T(e) can be reached by n_1. Similarly, because of having $outdegree(H(e)) > 1$, there is always at least one path form H(e) to n_0. Thus, we can conclude that G does not loss connective property when eliminating redundant arcs. □

Theorem 2. *In ddgraph $G = (N, A)$, with a complete path p (path from n_1 to n_0), we always reduce at least one arc $e \in p$.*

Proof. let $P = (e_1, e_2, ...e_q)$ is a complete path in ddgraph G, with $H(e_1) = n_1$ and $T(e_n) = n_0$. We assume that theorem is false. Hence, a path p has no redundant arc if any arc e of P; $indegree(T(e)) = 1$ or $outdegree(H(e)) = 1$. The arc e_1 has $H(e_1) = n_1$ so $outdegree(H(e_1)) = 1$. And because the G have no loop, we can infer that $indegree(T(e_1)) = 1$. Because of being a node of the ddgraph G, $T(e_1)$ satisfies the inequality $indegree(T(e_1)) + outdegree(T(e_1)) > 2$. So The inequality $outdegree(T(e_1)) > 1$ is true. Next, the edge e_2 has

$outdegree(H(e_2) = outdegree(T(e_1) > 1$. Likewise, the arc e_2 is not redundant arc, so $indegree(T(e_2)) = 1$. In summation, we can infer that $outdegree(T(e_2) > 1$. Similarly, by considering the other arc e_x of P (with $2 < x < q - 1$), $indegree(T(e_x)) = 1$ and $outdegree(T(e_x)) > 1$ are satisfied. At the arcs e_q where $T(e_q) = n_0$, we have $outdegree(H(e_q) > 1$ so there will be at least two arcs from the node $H(e_q)$. Because $indegree(n_0) = 1$, suppose that the arc e_y has $T(e_y) \neq n_0$. Ddgraph G has not cycles, therefore, we will always find at least one way to the n_0 which does not contain e_n from $T(e_y)$. As a result, $indegree(T(e_q)) > 1$ is true. In other words, it means that the arc e_q can be reduced and the theorem is true. □

3 Generating Simplified Ddgraph

In order to operate control flow testing, we need to perform two major steps which are selecting paths and generating test inputs. The proposed method does not focus on selecting paths in whole CFG. Instead, it selects paths in sub CFG of program. Hence, in order to get sub ddgraph and take advantage of black-box, the first step of the proposed method simplifies the original ddgraph. Then, in the second step, the method finds test paths covering branches of sub ddgraph, followed by generating set of a test inputs corresponding to these paths.

This section presents the first step which is interested in reducing all the redundant arcs covered by black-box. The procedure named SIMPLIFY which is shown in Algorithm 1 is to operate this step. Generally, the inputs of this procedure are the test input data generated by black-box and an original ddgraph. And the output of it is a sub ddgraph of the input ddgraph. To specify, classical CFG, first, is used for symbolic execution [7] to find paths corresponding to black box test inputs. Then, the list of visited arcs will be chosen from this paths. For example, in Fig. 2, with set of black-box's path $B = \{P_1, P_2, P_3\}$, where $P_1 = (e_1, e_2, e_5)$, $P_2 = (e_1, e_2, e_4, e_7, e_8)$, and $P_3 = (e_1, e_2, e_4, e_7, e_9)$, the visited list is extracted as follow: $V = \{e | \forall e \in P_1 \bigcup P_2 \bigcup P_3\}$, so $V = \{e_1, e_2, e_4, e_5, e_7, e_8, e_9\}$.

After having V, the redundant arcs are eliminated from ddgraph. To begin, if the visited list is empty then the process stops (line 1). Conversely, each arc will be removed from the visited list and the graph if it is a redundant arc (line 2 to 7). After eliminating redundant arcs from the ddgraph, the redundant nodes which have just one arc entering and leaving on it can appear in ddgraph G. In ADJUST1 procedure presented in Algorithm 2 (line 8), all this nodes are eliminated from graph. Then the arcs binding to the redundant nodes are bound to create new arcs in both graph and visited list (line 8). Lastly, the visited list is checked to remove arcs not existing in ddgraph (line 9 to 13). After all, the process above is repeated until the visited list is empty.

When an arc is eliminated the number of outcomes in its head node and the number of incomes in its tail node will decrease by one. Hence, after that some arcs are not redundant anymore. For example, in Fig. 2(a), there are two redundant arcs from node 8 to node 1 but after one of them is removed the other is not redundant.

Algorithm 1. SIMPLIFY procedure

Input: Ddgraph G, set of visited arcs
Output: Ddgraph G' with smaller size

```
1: while visited_list is empty do
2:   for all e in visited_list do
3:     if indegree(T(e)) > 1&&outdegree(H(e)) > 1 then
4:       A = A − e
5:       visited_list = visisted_list-e
6:     end if
7:   end for
8:   ADJUST1(G, visited_list)
9:   for all e in visited_list do
10:    if e ∄ G then
11:      visited_list = visited_list − e
12:    end if
13:  end for
14: end while
```

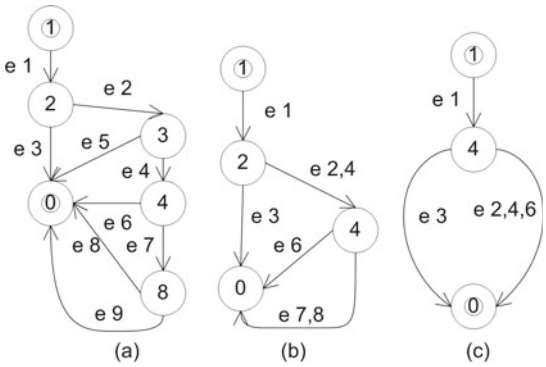

Fig. 2. Simplifying process of the ddgraph.

Figure 2 shows ddgraph in different status in three rounds of simplifying process corresponding to visited list V. In more detail, with the redundant arcs e_5, e_8, and e_9 in first step, the arcs e_5, e_9, nodes 3 and 8 are reduced to generate ddgraph in Fig. 2(b). Then, the visited list V remains three arcs $\{e_1, e_{2,4}, e_{7,8}\}$. In second round, after eliminating the arc $e_{7,8}$, the visited list V consists of two arcs e_1 and $e_{2,4}$. However the arc $e_{2,4}$ does not exist in the ddgraph in Fig. 2(c), thus it is removed. The V remains one arc e_1 which is not a redundant arc thus the reducing process exits.

4 Generating Test Input Data

With the sub ddgraph, the working space of method was significantly declined. Then, in this section, the method focuses on selecting non duplicated paths and generating the set of test input corresponding to selected paths.

Algorithm 2. ADJUST1 procedure

Input: graph G, set of arcs
Output: DGraph G', set of arcs correspond to G'

```
1: for all nodes n in N do
2:    if indegree(n)=1 && outdegree(n)=1 then
3:       for all e ∈ visited_list do
4:          if T(e) = n then
5:             e_i = e
6:          end if
7:          if H(e) = n then
8:             e_j = e
9:          end if
10:      end for
11:      e_new = (H(e_i), T(e_j))
12:      visited_list = (visited_list - e_j - e_i + e_new)
13:      for all e in A do
14:         if T(e) = n then
15:            e_i = e
16:         end if
17:         if H(e) = n then
18:            e_j = e
19:         end if
20:      end for
21:      e_new = (H(e_i), T(e_j))
22:      A = (A - e_j - e_i + e_new)
23:      N = (N - n)
24:   end if
25: end for
```

According to the Theorem 2 described in Sect. 2, a complete path always has at least one redundant arc. Thus, after finding complete paths, at least one arc may be reduced to create smaller ddgraph. In the proposed method, the selecting process continually finds complete paths and reduces redundant arcs and nodes until the ddgraph has only one arc.

The selecting process is presented in the procedure named FIND_BASIC which is shown in Algorithm 3. The following is a more detailed presentation of the method to select test paths. Initially, a set of return paths is created (line 1). Then, a path with start arc is created. If the generated path is complete (line 8), it means that the ddgraph has just one arc, then the path is added to the set of return paths (line 9) and the finding process ends (line 10). Otherwise, it is added to the stack (line 14). From line 15 to 32 is the deep-first-search process with an alteration. All redundant arcs in each found complete path are removed from the ddgraph (line 21 to 24). After the reduction, the graph is adjusted with the redundant eliminated nodes and the arcs which are bound together in ADJUST2 procedure presented in Algorithm 4 (line 26).

Algorithm 3. FIND_BASIC procedure

Input: Ddgraph G
Output: Set of complete paths covers all branches of G

```
1: Create array of paths A
2: while true do
3:    Create s stack S
4:    for all arcs e in G do
5:       if H(e) = n_1 then
6:          create a path p
7:          add e to p
8:          if p is complete path then
9:             add p to A
10:            return A
11:         end if
12:      end if
13:   end for
14:   S.pop(p)
15:   while S is empty do
16:      for all arcs e in G do
17:         if e is adjacent p then
18:            add e to p
19:            if p is complete path then
20:               add p to A
21:               for all arcs e in p do
22:                  if indegree(T(e)) > 1&&outdegree(H(e)) > 1 then
23:                     A = A − e
24:                  end if
25:               end for
26:               ADJUST2(G)
27:            else
28:               S.push(p)
29:            end if
30:         end if
31:      end for
32:   end while
33: end while
```

Figure 3 shows a ddgraph in the finding process which is implemented to generate a set of path covering all branches of it. For instance, in Fig. 3(a), after finding path $P_1 = (e_1, e_3, e_7)$, arc e_7 and node 4 are reduced. Then, in step 2, arc $e_{3,6}$ and node 2 are eliminated in path $P_2 = (e_1, e_3, e_6)$. Finally, after reducing arc e_4 and with ddgraph having only one arc $e_{1,2,5}$, the finding process is done. The process returns set of paths $S = \{(e_1, e_3, e_7), (e_1, e_3, e_6), (e_1, e_2, e_4), (e_1, e_2, e_5)\}$ covering all branches of the ddgraph.

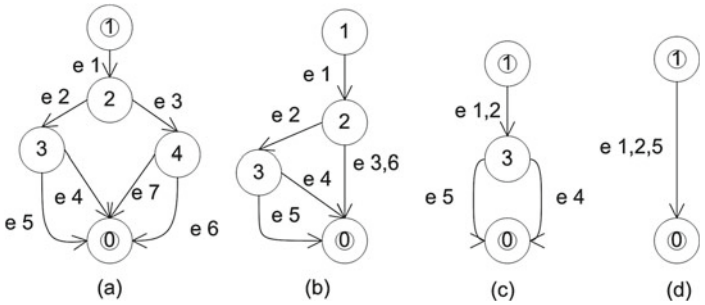

Fig. 3. Finding process.

5 Time Complexity

In this section, we estimate the time complexity of the proposed method. It can be calculated based on the time complexity of two major Algorithms 1 and 3. The time complexity of the method depends on many factors, such as the number of decision nodes, the shape of graph, and the coverage of black-box test cases. In the worst cases, if simplifying process is not operated, it may be approximately $O(n^4)$. In almost cases, when combining black-box, the time complexity of the proposed method is approximately $O(n^2log(n))$. The following explains how the proposed method can reach time complexity $O(n^2log(n))$.

At first glance, the time complexity of the proposed method is the maximum of the simplifying process's time complexity and the selecting process's one. So, the time complexity of generating simplified ddgraph and the one of generating simplified ddgraph are considered, respectively. First, the generating simplified ddgraph is presented in Algorithm 1. In the while loop of Algorithm 1, we can easily see that the codes from line 2 to line 7 which reduce redundant arcs have time complexity O(n). And, O(n) is the time complexity of codes eliminating not existed arcs from line 9 to 13. ADJUST1 procedure presented in Algorithm 2 has two loops. Thus, ADJUST1's time complexity is approximately $O(n^2)$. The SIMPLIFY procedure's time complexity depends on how many arcs in the visited list. In practice, when black-box's path covers all branches of the ddgraph, the SIMPLIFY procedure is not operated. But in order to estimate time complexity, we can assume that the SIMPLIFY procedure is performed when all branches are visited. So, in the worst case, in Fig. 1(c), when ddgraph has shape like complete binary tree, in each round we can reduce 1 from the height of tree when half of arcs which are connected to ending node are removed. Height H of tree is calculated as follows: $H = log(|N| - 1) + 1$. So we have time complexity of SIMPLIFY procedure is approximately $O(n^2log(n))$.

In the FIND_BASIC procedure presented in Algorithm 3, the time complexity depends on how many arcs can be reduced from the path found by DFS. With the best case, the ddgraph which has series of condition nodes, the FIND_BASIC procedure is performed just one time. On the contrary, in the worst case, with ddgraph such as Fig. 1(c), we can reduce one arc from a found path and the loop

Algorithm 4. ADJUST2 procedure

Input: graph G
Output: Ddgraph G'

```
1: for all nodes n in N do
2:   if indegree(n)=1 && outdegree(n)=1) then
3:     for all e in A do
4:       if T(e) = n then
5:         e_i = e
6:       end if
7:       if H(e) = n then
8:         e_j = e
9:       end if
10:    end for
11:    e_new = (H(e_i), T(e_j))
12:    A = (A - e_j - e_i + e_new)
13:    N = (N - n)
14:  end if
15: end for
```

Table 1. Black-box test input paths.

No.	Usage	Member score	Date	Path
1	0	50	5	e_1, e_3, e_{19}
2	0	150	25	e_1, e_3, e_{18}
3	100	50	5	e_1, e_2, e_5, e_6
4	400	150	25	e_1, e_2, e_5, e_7, e_8
5	600	50	5	$e_1, e_2, e_4, e_{11}, e_{13}, e_{17}$
6	600	50	25	$e_1, e_2, e_4, e_{11}, e_{12}, e_{15}$
7	600	150	5	$e_1, e_2, e_4, e_{11}, e_{13}, e_{16}$
8	600	150	25	$e_1, e_2, e_4, e_{11}, e_{12}, e_{15}$
9	600	300	5	$e_1, e_2, e_4, e_{11}, e_{13}, e_{16}$
10	600	300	25	$e_1, e_2, e_4, e_{11}, e_{12}, e_{14}$

will be repeated $|N| - 1$ times. Furthermore, depth-first searching algorithm is executed repeatedly. In each Depth-first searching loop, we use ADJUST2 produre so that time complexity can be calculated as $O(|A|) * O(|N| * |A|)$, approximate $O(n^3)$. Hence, when being performed just one time in the best case, it has $O(n^3)$ time complexity. And in the worst case, it has $O(n^4)$ time complexity.

Although the FIND_BASIC procedure needs a lot of time but after being simplified, the size of ddgraph is very small. We can examine the example of calculateBill method in java. The source code, CFG, and ddgraph of calculateBill is available at[1]. This method implements calculating bill task of restaurant. The input of this method are usage, member's score and date. The Table 1 shows that

[1] http://www.uet.vnu.edu.vn/~hungpn/calculateBill.jpg/

Table 2. While-box test inputs.

No.	Path	Constraints	Test input
1	(e_1, e_2, e_4, e_{10})	$Usage > 0$	$Usage = 500.0$
		$Usage \geq 500$	
		$((40) + 50 + Usage) \leq 600$	
2	$(e_1, e_2, e_5, e_7, e_9)$	$Usage > 0$	$Usage = 201.0$
		$Usage < 500$	
		$Usage > 200$	
		$((40) + 50 + Usage + (Usage - 50) * 0.1) < 400$	

10 sets of black-box test inputs covering almost of branches of the ddgraph. As a result, the ddgraph of this method was declined from graph having 11 nodes, 19 arcs to graph having 3 nodes and 3 arcs. Thus, white-box has to find two paths $S = \{(e_1, e_2, e_4, e_{10}), (e_1, e_2, e_5, e_7, e_9)\}$ for covering all branches of the sub graph. In summary, the final result is that the complexity time of whole process is approximately $O(n^2 log(n))$. The Table 2 shows the result of the while-box's test inputs corresponding with the test paths and constraints of them.

6 Conclusion

As show above, with all black-box's test paths eliminated, the control flow testing significantly reduces complexity. The time complexity of this method declines from $O(n^4)$ when performing white-box separately to $O(n^2 log(n))$ when combining with black-box. In addition, by continually removing the redundant arcs and nodes after finding complete paths, the proposed method can generate set of test inputs, which followed by yielding non duplicated paths. As a result, by take full advantage of black-box result, this method can select a small set of test inputs that cover all branches of graph with little effort. Does this method not only propose an approach get most of the black-box, it is also completely operated to reuse test cases in the context of program evolution.

Albeit, the proposed method has not supported for program unit with loop yet but in future, for solving this problem, we will focus on detecting and separating loop from graph to present it under no loop graph form. Now, this method is being implemented as a plugin of Eclipse for control flow testing java unit. In the next time, we will complete the plugin for evaluating performance of this method in practice.

References

1. McCabe, T., Thomas, J.: Structural testing: a software testing methodology using the cyclomatic complexity metric. NIST Special Publication 500–99, Washington, D.C. (1982)

2. Bertolino, A., Marre, M.: Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Trans. Softw. Eng. **20**(12), 885–899 (1994)
3. Guangmei, Z., Rui, C., Xiaowei, L., Congying, H.: The automatic generation of basis set of path for path testing. In: Proceedings of the 14th Asian Test Symposium (ATS'05) (2005)
4. Marre, M., Bertolino, A.: Using spanning sets for coverage testing. IEEE Trans. Softw. Eng. **29**(11), 974–984 (1993)
5. Gupta, R., Soffa, M.L.: Employing static information in the generation of test cases. Softw. Test. Verification Reliab. **3**(1), 29–48 (1993)
6. Ghiduk, A.S., Said, O., Aljahdali, S.: Basis test paths generation using genetic algorithm. In: International Conference on Computing The Information Technology (ICCIT), pp. 303–308 (2012)
7. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)