Mutation Testing Using Symbolic Execution and Path

Khalid A. Buragga[1], Sultan Aljahdali[2], Marcel Karam[3], Ahmed S. Ghiduk[2,4]

[1]College of Computing and Information Technology, Northern Border University, Saudi Arabia
E-mail: nbu10@nbu.edu.sa
[2]College of Computers and Information Technology, Taif University, Saudi Arabia
Email: aljahdali@tu.edu.sa
[3]Department of Computer Science, American University of Beirut
Email: marcel.karam@aub.edu.lb
[4]Department of Mathematics and Computer Science, Faculty of Science,
Beni-Suef University, Egypt
Email: asaghiduk@tu.edu.sa

Abstract. In this paper, we introduce a new technique for generating set of test data for mutation testing. This technique automatically generates a set of program paths that satisfy the branch coverage criterion by implementing a path generation algorithm proposed by Bertolino and Marré. Then, the proposed technique symbolically executes the generated set of paths to create a system of branch conditions for each path and solve this set of conditions to find the required set of test data for killing the mutants of the program under test. The technique determines infeasible paths by checking the consistency of each system of conditions. For each infeasible path, the technique finds, if possible, a new feasible path. The paper also presents the results of experiments that have been carried out to evaluate the effectiveness of the proposed technique.

Keywords: mutation testing, symbolic execution, test path, test path generation

1 Introduction

Software testing has two main aspects: test generation and application of a test data adequacy criterion. A test generation technique is an algorithm that generates test cases, whereas an adequacy criterion is a predicate that determines whether the testing process is finished [1]. Several test data adequacy criteria have been proposed, such as control flow-based and data flow-based criteria. One of the major difficulties in software testing is the automatic generation of test data that satisfy a given criterion.

Symbolic execution can be used to generate test data for a selected adequacy criterion. It involves executing a program using symbolic values of variables instead of numeric values, and requires the determination of program paths that are to be followed in order to satisfy the selected adequacy criterion. So, the symbolic execution testing system should incorporate a path selection strategy in which the expressions produced by the symbolic execution are used to identify the required paths. Several symbolic execution systems have been built [2-9, 21]. A symbolic execution system has been developed by Girgis [10, 11] that automatically generates a subset of program paths according to a certain control flow criterion. This subset is called the ZOT-subset, since it requires paths that traverse loops zero, one and two times. The paths of this subset are presented to the user to identify feasible paths, then the system selects feasible paths from the ZOT-subset that cover a data flow path selection criterion.

The main contributions of this paper are: introducing a technique for generating set of test inputs for killing the program mutants by symbolically executing a generated set of program paths that satisfy the branch coverage criterion. Then, the proposed technique create a system of branch conditions for each path and solve this set of conditions to find the required set of test data for killing the mutants of the program under test. The technique checks path infeasibility by checking the consistency of each system of conditions and finds, if possible, a new feasible path.

The paper is organized as follows: Section 2 describes the path generation algorithm of our technique and some basic testing concepts. Section 3 describes the components of our technique. Section 4 presents the results of experiments that have been carried out to evaluate the effectiveness of the technique.

2 Background

This section describes the path generation algorithm FTPS implemented in our system [12] and discuss the ideas of path generation and mutation testing.

2.1 The Path Generation Algorithm

The control flow of a program is represented by a directed graph, called flow-graph. A directed graph or digraph G = (V, E) consists of a set V of nodes or vertices, and a set E 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. If H(e) = T(é), e and é are called adjacent arcs. For a node n in V, indegree(n) is the number of arcs entering it, and outdegree(n) is the number of arcs leaving it. Figure 2 shows the flow-graph for the example program of Figure 1.

A path P of length q in a digraph G is a sequence of edges $P = e_1, e_2, e_3, \ldots, e_q$, where $T(e_{i+1}) = H(e_i)$ for i = 1 , 2, …, q-1. P is said to be a path from e_1 to e_q.

The FTPS algorithm uses a flow-graph representation called ddgraph (decision-to-decision graph), which is particularly suitable for the purposes of branch testing. A ddgraph, as defined in [12], is a digraph G =(V, E) with unique entry arc e_0 and unique exit arc e_k, such that for each node $n \in V$, $n \neq T(e_0)$, $n \neq H(e_k)$, (indegree(n) + outdegree(n)) > 2, while indegree($T(e_0)$) = 0 and outdgree($T(e_0)$) = 1, indegree($H(e_k)$) = 1 and outdgree($H(e_k)$) = 0. The arcs of a ddgraph represent branches of a program, where a branch is a strictly sequential set of program statements uninterrupted by either decisions or junctions. Figure 3 shows the ddgraph G_1 that corresponds to the flow-graph of Figure 2. It should be noted that two auxiliary arcs, e_0 and e_9, have been added to G_1 as entry and exit arcs.

The algorithm uses two relations between the dd-graph arcs, namely, the dominance and implication relations. Let G = (V, E) be a dd-graph with unique entry arc e_0 and unique exit arc e_k. The *dominance* relation between two arcs in G is defined as follows: An arc e_i dominates an arc e_j if every path P from the entry arc e_0 to e_j contains e_i. The *implication* relation between two arcs in G is defined as follows: An arc e_i implies an arc e_j if every path P from e_i to the exit arc e_k contains e_j.

By applying the dominance relation between the arcs of G, a tree (whose nodes represent the ddgraph arcs) rooted at e_0, can be obtained. This is called the *dominator tree* DT(G). By applying the implication relation between the arcs of G, a tree (whose nodes represent the ddgraph arcs) rooted at e_k, can be obtained. This is called the *implied tree* IT(G). Figure 5 shows the implied tree of the ddgraph G_1, $IT(G_1)$. In this figure, the sequence of arcs $P_{IT} = e_1, e_4, e_6, e_9$ is an implication path in $IT(G_1)$.

The FTPS algorithm constructs ddgraph paths as follows: it first derives a dominance or an implication path, and then fills possible discontinuities with a 'suitable' path in G. For example, in the path $P_{DT} = e_0, e_4, e_5, e_7$ on $DT(G_1)$, the discontinuity between e_0 and e_4 may be filled with the path $P' = e_2$, obtaining the ddgraph path $P = e_0, e_2, e_4, e_5, e_7$. A set of paths $\wp = \{P_1, \ldots, P_n\}$ is a path cover for a ddgraph G = (V, E) if for each arc $e \in E$ there exists at least one path in $\wp$ containing e. For example, the set of paths $\wp = \{P_1, P_2, P_3, P_4\}$ is a path cover for the ddgraph G_1 of Figure 3, where:

$P_1 = e_0, e_2, e_3, e_4, e_6, e_9$; $P_2 = e_0, e_1, e_4, e_6, e_9$;
$P_3 = e_0, e_1, e_4, e_5, e_8, e_6, e_9$; $P_4 = e_0, e_1, e_4, e_5, e_7, e_6, e_9$.

```
St. No.                              Basic Block
 1     INTEGER N,D,R,T,Q                    1
 2     READ*, N, D
 3     IF (D.LT.N) THEN
 4       I = N                              2
 5       N = D
 6       D = I
 7     END IF                               3
 8     Q = 0
 9     R = N
10     T = D
11       10     IF (R.GE.T) THEN 4
12       T = T * 2                          5
13       GO TO 10
14     END IF                               6
15      20      IF (T.NE.D) THEN 7
16       Q = Q * 2                          8
17       T = T / 2
18       IF (T.LE.R) THEN
19         R = R - T                        9
20         Q = Q + 1
21       END IF                  10
22       GO TO 20
23     END IF                               11
24     PRINT*, Q, R
25     END                                  12
```

Fig. 1. Example program

To find a path cover on a ddgraph, Bertolino and Mareé introduced the notion of unconstrained arcs. The set UE(G) of unconstrained arcs of G can be obtained as

$$UE(G) = DTL(G) \cap ITL(G) \qquad (1)$$

where DTL(G) is the set of leaves of DT(G) and ITL(G) is the set of leaves of IT(G).

For the ddgraph G1 of Figure 3, the set of unconstrained arcs is:

$$UE(G_1) = DTL(G_1) \cap ITL(G_1) = \{e_1, e_2, e_3, e_7, e_8\}$$

where: $DTL(G_1) = \{e_1, e_2, e_3, e_7, e_8, e_9\}$ and $ITL(G_1) = \{e_0, e_1, e_2, e_3, e_5, e_7, e_8\}$.

Different path covers can be derived by the algorithm FTPS by implementing different selection policies. Two selection policies are considered: the *minimum-number-of-paths* policy, which aims at reducing the number of paths, and the *less-predicates* policy which aims at preventing the generation of infeasible paths.

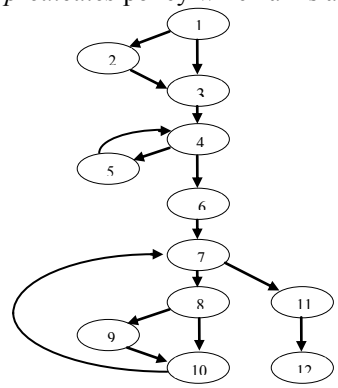

Fig. 2. The flow-graph for the example program.

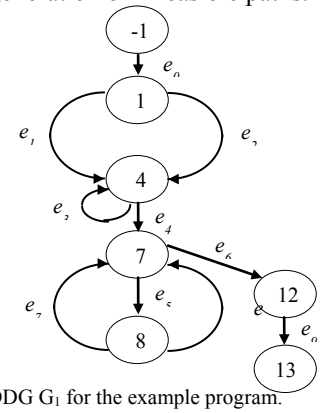

Fig. 3. DDG G_1 for the example program.

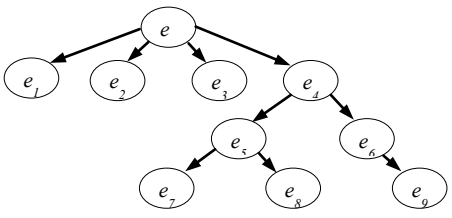

Fig. 4. The dominator tree of G1, DT(G1).

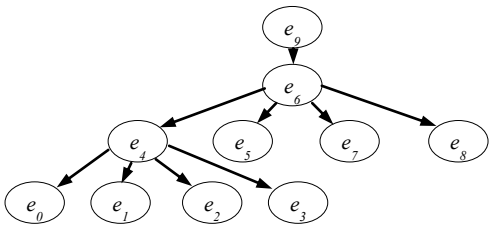

Fig. 5. The implied tree of G1, IT(G1).

2.2 Mutation testing

In mutation testing, a set of faulty programs p', called mutants, is generated by seeding faults into the original program p. A mutant is generated by making a single small change to the original program. For example, **Error! Reference source not found.** shows a first-order mutant in the mutated program p' generated by changing the *and* (&&) operator in the original program p into the *or* (||) operator in the mutated p'. In addition, **Error! Reference source not found.** gives a second-order mutant by changing two operators (&&) and (>) in p into (||) and (<) in p'. A transformation rule that generates a mutant from the original program is known as a mutation operator [14].

Table 1. An Example of Mutation Operation

Original Program p	Mutated Program p'	
	First Order Mutant	Second Order Mutant
if (a >0&& b> 0)	if (a>0 ‖ b > 0)	if (a > 0 ‖ b < 0)

Each mutated program p' will be executed using a test set T. If the result of running p' is different from the result of running the original program p for any test case in T (i.e., p'(t) ≠ p(t) for any t of T), then the mutated program p' is said to be "killed", otherwise it is said to have "survived". The adequacy level of the test set T can be measured by a mutation score [15] that is computed in terms of the number of mutants killed by T as follows.

$$MS(P,T) = \frac{\#\ of\ killed\ \ Mutants}{Total\ \ no.\ of\ Mutants - no.of\ Equivalent\ \ \ Mutants} \qquad (2)$$

The aim of mutation testing is finding test set T [16, 17].

2.3 Test-Coverage Criteria

A test-coverage criterion is used to determine whether a program has been adequately tested. It specifies a set of program entities that must be exercised by the test cases on which the program is executed during the testing process.

2.3.1 Control-Flow Criteria

The most test coverage criteria included in this category are:

- All-nodes test coverage criterion requires that each node in the control-flow graph be executed by some test case. Therefore, it is also called statement testing.
- All-edges test coverage criterion requires that each edge in the control-flow graph be traversed by some test case during some program execution. This form of testing is also called branch testing.
- All-paths criterion requires that every complete path (i.e., a path from the entry node to the exit node of the control-flow graph) in the program be tested. This form of testing is also called path testing.

2.3.2 Data-Flow Criteria

Given a set of test cases, let Q be the set of complete paths exercised by the program executions for these test cases. For each of the data flow based test-coverage criterion, the conditions that Q must meet for the test criterion to be satisfied are given below:

- All-defs is satisfied if Q includes a def-clear path from every definition to some corresponding use (c-use or p-use).
- All-c-uses is satisfied if Q includes a def-clear path from every definition to all of its corresponding c-uses.
- All-p-uses is satisfied if Q includes a def-clear path from every definition to all of its corresponding p-uses.
- All-uses is satisfied if Q includes a def-clear path from every definition to each of its both c-uses and p-uses.
- All-du-paths is satisfied if Q includes all du-paths for each definition. Therefore if there are multiple paths between a given definition and a use, they must all be included.

3 The Proposed Technique

This section describes our symbolic execution based mutation testing technique. Figure 6 gives the block diagram of the proposed technique. It performs the following actions:

1. analysis and reformatting of source code of the program under test (PUT).
2. generating a set of program paths that satisfy the branch coverage criterion.
3. symbolically executing the generated paths and creating a system of conditions for each paths.
4. checking the feasibility of the generated paths by checking the consistency of the system of conditions.
5. finding a set of test inputs for each feasible path by solving the generated set of conditions.
6. running the tested program and its mutants using the generated test inputs and finding the killed mutants.

In the following subsections, the components of the technique are described further.

3.1 Analysis phase

The analysis and reformatting module is an adapted version of the testing system proposed by Girgis and Woodward [18]. This module classifies program statements and reformats some of them to facilitate the construction of the program flow graph. A file is produced, which contains the reformatted version of the source code, and this is passed to the path generation module. This module seeds the original program with errors and generates set of mutants of the program under test (PUT).

3.2 Path generation phase

This module performs the following actions in order to generate a set of program paths that satisfy the branch coverage criterion.

1. Constructing the control flow graph of the PUT.
2. Forming the ddgraph of the PUT (see section 2.1).
3. Finding for each arc in the ddgraph the set of its dominance arcs and the set of its implication arcs by using the algorithm given in [19]. Using these sets of arcs, the dominator tree DT(G) and the implied tree IT(G) of the ddgraph are built. Then, the set of unconstrained arcs, UE(G), of the given program is found from the dominator tree and the implied tree by using eq. (1), as described in section 2.1.
4. Building a path cover $\wp$ using the FTPS algorithm, as described in section 2.1. The paths in $\wp$ are derived one at a time. To construct each path, FTPS selects an as yet uncovered unconstrained arc e_u, using the chosen path selection policy, the min-no-of-paths or the less-pred, and then finds a path from entry arc e_0 to the exit arc e_k, using arc e_u. This procedure is repeated until all unconstrained arcs are covered.

At the end of this phase, a subset of program paths are generated that cover all the edges of the program ddgraph.

3.3 Symbolic execution phase

In this phase, the generated paths in the second phase are symbolically executed, and a set of branch conditions is created for each path, as described in [10]. During the symbolic execution of a path, the system assigns a symbolic value to each input variable. These symbolic values are supplied from the list (A1, A2, A3,..). For example, the symbolic values assigned to the input variables N and D of the example

program are A1 and A2, respectively. So, the branch conditions created will be in terms of these symbolic input values. A symbol table is used to keep track of the symbolic values and the states of program variables during the symbolic execution of a path. The state of each variable is monitored in order to detect the presence of any data flow anomalies. Note that, during program execution, a variable could be in one of four possible states: undefined (state U), defined (state D), defined and then referenced (state R), anomalous (states UR, DU or DD).

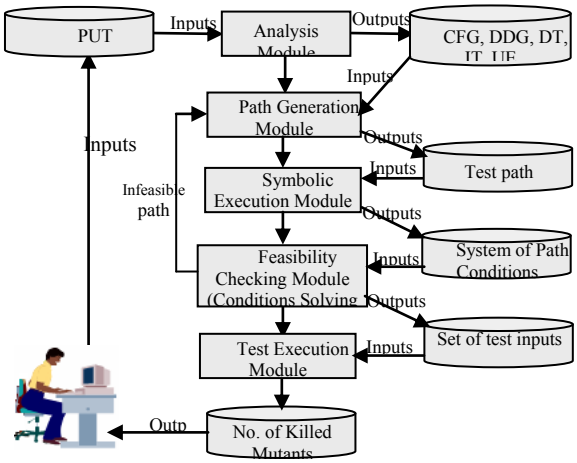

Fig. 6. The block diagram of the proposed technique.

3.4 Feasibility checking

This module checks the consistency of the generated set of branch conditions. This set of branch conditions created for any path forms a system of equalities and inequalities. This system describes the subset of the input domain that causes the path to be executed. Solving the systems of conditions of the generated paths gives test data that cause the paths to be executed, and hence fulfils the branch coverage criterion.

For each possible policy used for the generation of the paths, infeasible paths may be chosen, and so the paths cannot actually satisfy the branch testing criterion.

To cope with this problem, the path generation and the symbolic execution modules have been merged. The actions performed by the merged modules are as follows:

1. The path generation module finds a path P_u that covers an unconstrained arc e_u.
2. The symbolic execution module symbolically executes P_u, and creates a system of branch conditions for it.
3. The system presents the conditions to the feasibility checking module to check their consistency. If the system of conditions of P_u is consistent (i.e. soluble), the path is feasible. In this case, the system adds P_u to $\wp$, then selects an as yet uncovered unconstrained arc e_u, and goes to step 1. On the other hand, if the system of conditions of P_u is inconsistent, the path is infeasible and needs to be replaced. Information about the combination of arcs that form this infeasible path is stored in a table. This information can be used in the path generation phase to prevent the future construction of paths containing these (infeasible) combinations of arcs.
4. The system constructs a new path to cover arc e_u as follows: Let S be a subpath in G, S = e_0, … , e_u, … , e_k, where e_0, … , e_u is the dominance path in DT(G) between e_0 and e_u, and e_u, … , e_k is the implication path in IT(G) between e_u and e_k. By filling in any possible way the discontinuities in S, the set of all paths from

e_0 to e_k and containing e_u, can be obtained. So, to obtain a new path through e_u, the system fills the discontinuities in S with new subpaths, taking into account that information about the infeasibility of P_u that has been saved [12].

5. Then, the new path is given to the symbolic execution module and steps 2 through 4 are repeated until a feasible path that covers arc e_u is found. It should be noted that it is possible to find an infeasible path that cannot be replaced with a feasible one.
6. Steps 1 through 5 are repeated until a path cover $\wp$ is obtained that consists only of feasible paths.

During the generation of a path, the system accesses the table of impossible combinations of arcs to check its infeasibility. In particular, SELECT_AN_ARC can select the next unconstrained arc from a smaller subset, obtained by eliminating from the set UE all those arcs recognized (and stored in the table) as impossible to combine with the arc already chosen in the path being constructed.

At the completion of the path generation process, a report is produced containing the unconstrained arcs covered by each generated path, in addition to the ones that cannot be covered by any feasible path, if any.

In this way a path cover $\wp$ containing feasible paths that fulfil the branch coverage criterion, and a system of branch conditions for each one of these paths, are obtained. The final task is to solve the systems of conditions in order to generate test data to cover all the branches of the program.

3.5 Test execution module

This module executes the original program and its mutants using the test inputs which are yielding from solving the consistency systems of conditions for each feasible path. Then, this module finds the killed mutants.

4 The Experiments

This section describe the experiments that have been carried out in order to evaluate the error-exposing ability of the system and the data generated by applying the two arc selection policies, the min-no-of-paths and the less-pred, used by the path generation algorithm.

In these experiments, nine programs were selected and seeded with errors one at a time. In each case, the erroneous program was analysed and two path cover sets were generated by applying the two arc selection policies. Then, the paths of the two sets were symbolically executed by the system, and systems of inequalities were produced. The erroneous program was executed with the data generated by solving the inequalities. The output of this execution was compared with the correct output, which was obtained by executing the correct program with the same data. The success of the system in discovering the error is judged by:

1. the appearance of data-flow anomaly messages during the symbolic execution of the erroneous program.
2. the occurrence of any change in the form of the inequalities or their consistency (i.e. consistent inequalities of the correct program become inconsistent, and vice versa).
3. the generation of different symbolic output from symbolically executing the same path in the correct program and in the erroneous one.
4. the occurrence of a deviation in the actual output.

The errors that were seeded into programs in these experiments fall into two categories: domain errors and computation errors. The definitions of these two

categories are given by [20]. Table 2 shows these errors and their frequencies in the experiments.

The results of the experiments were analyzed. The effectiveness of the system with each arc selection policy was studied with respect to all the seeded errors, all the seeded errors of each category, and all the seeded errors of each type.

The results of the experiments showed that:

- with min-no-of-paths policy, 89 out of 101 errors were discovered, which represents 88% of all seeded errors. The undiscovered errors were: 2 of type D1, 1 of type D2, 2 of type D3, 3 of type D4, 1 of type C2, and 3 of type C3.
- with less-pred policy, 92 out of 101 errors were discovered, which represents 91% of all seeded errors. The undiscovered errors were: 2 of type D3, 2 of type D4, 3 of type C3, 1 of type C2, 1 of type C5.

Table 2. The types of seeded errors and their frequencies

Code	Error type	Error Frequency
C	**Computation errors**	
C1	wrong variable definition	5
C2	wrong arithmetic operator	13
C3	wrong variable reference	23
C4	incorrect constant value	10
C5	statement wrongly placed	5
C6	missing computation	7
C7	a variable replaced by a constant	3
		66
D	**Domain errors**	
D1	wrong relational operator	12
D2	a variable replaced by a constant	5
D3	wrong variable reference	9
D4	incorrect constant value	9
		35
		101

These results indicate that the path covers generated by using the less-pred policy have higher ability of discovering errors than those generated by using the min-no-of-paths policy.

Tables 3 and 4 show the discovered errors during executing of the proposed technique. Some of the seeded errors were discovered during the symbolic execution of the erroneous program by the data flow anomaly messages, the generation of inconsistent (consistent) inequalities which should be consistent (inconsistent), or the generation of different symbolic output for the same path in the correct program and in the erroneous one. Others were discovered by comparing the results of executing the erroneous program and the correct one with the same data, which were generated by solving the systems of inequalities produced by the system for the erroneous program. We can see that some errors were discovered by more than one of the above methods.

Table 3. Number of Discovered Errors by Minimum Number of Paths Criterion.

	Comparing Results	Data Flow Anomaly	Comparing Inequalities	Comparing Symbolic Output	Symbolic Execution	No. of Discovered Errors
Computation Errors	35	20	20	50	63	63
Domain Errors	17	1	21	1	21	26
Total	52	21	41	51	84	89

From these tables, it can be seen that most of the errors were discovered during the symbolic execution of the erroneous programs along the paths generated by applying both arc selection policies.

Figure 7 shows the percentage of computation and domain errors discovered using the path cover generated by applying the min-no-of-paths policy and the less-pred policy. By comparing the percentage of errors discovered, it can be seen that the path cover of the min-no-of-paths policy has discovered more computation errors than that

of the less-pred policy, while the path cover of the less-pred policy has discovered more domain errors than that of the min-no-of-paths policy.

Table 4. Number of Discovered Errors by Less-Predicates Criterion.

	Comparing Results	Data Flow Anomaly	Comparing Inequalities	Comparing Symbolic Output	Symbolic Execution	No. of Discovered Errors
Computation Errors	38	21	20	50	61	61
Domain Errors	24	1	24	1	24	31
Total	62	22	44	51	85	92

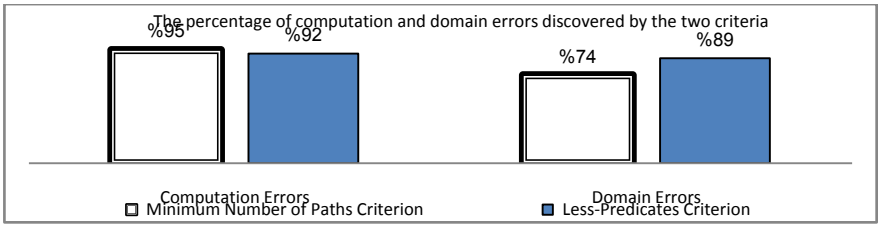

Fig. 7. The percentage of computation and domain errors discovered by the min-no-of-paths policy and the less-pred policy.

Table 5. Number and percentage of errors of each type discovered with the path cover generated by applying the two arc selection policies.

	Less-Pred Policy				Min-No-of-Paths Policy			
Error Type	Comparing Results		Symbolic Execution		Comparing Results		Symbolic Execution	
C1	5	100%	5	100%	5	100%	5	100%
C2	6	46.2%	12	92.3%	6	46.2%	12	92.3%
C3	12	52.2%	20	87%	11	47.8%	21	91.3%
C4	7	70%	10	100%	5	50%	10	100%
C5	3	60%	4	80%	3	60%	5	100%
C6	3	42.9%	7	100%	3	42.9%	7	100%
C7	2	66.7%	3	100%	2	66.7%	3	100%
D1	11	91.7%	9	75%	8	66.7%	8	66.7%
D2	4	80%	5	100%	3	60%	4	80%
D3	3	33.3%	5	55.6%	3	33.3%	5	55.6%
D4	6	66.7%	5	55.6%	3	33.3%	4	44.4%

Tables 5 shows the number and percentage of errors of each type discovered either during symbolic execution or by comparing results, using the path cover generated by applying the two arc selection policies. It can be seen from Table 5 that, with both policies, most of the errors of type C2, C3, C4, C5, C6, C7, D2 and D3 were discovered during the symbolic execution, and the errors of type C1 were discovered during the symbolic execution and by comparing results with the same percentage. The two policies differ in the ability of discovering the errors of type D1 and D4. With the less-pred policy, comparing results discovered more errors of these types than the symbolic execution. With the min-no-of-paths policy, the errors of type D1 were discovered during the symbolic execution and by comparing results with the same percentage, but the symbolic execution discovered more errors of type D4 than comparing results.

5 Conclusions

The paper presented an empirical study of the use of symbolic execution to aid the generation of test data for mutation testing. A symbolic execution system has been developed to automatically generate a set of program paths that satisfy the branch coverage criterion by implementing the path selection algorithm (FTPS) proposed by [12]. The system symbolically executes the generated set of paths and creates a system of branch conditions for each one. By solving the systems of conditions of the feasible paths one can obtain test data for mutation testing.

Experiments have been carried out to evaluate the error-exposing ability of the system. The results showed that the path covers generated by using the less-pred policy have higher ability of discovering errors than those generated by using the min-

no-of-paths policy. The results also showed that most of the errors were discovered during the symbolic execution of the erroneous programs along the paths generated by applying both arc selection policies. More empirical studies will do in the future work to compare the proposed technique with other mutation testing techniques.

References

1. P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering, 19(8), 774-787, 1993.
2. R. S. Boyer, B. Elspas and K. N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable software, 234-245, 1975.
3. L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 2(3), 215-222, 1976.
4. J. C. King. Symbolic execution and program testing. Communications of the ACM, 19 (7), 385-394, 1976.
5. W. E. Howden. Symbolic testing and the DISSECT Symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4), 266-278. 1977.
6. D. Hedley and M. A. Hennell. The causes and effects of infeasible paths in computer programs. Proceedings of Eighth International Conference on Software Engineering, IEEE Computer Society, 259-266. 1985.
7. T. E. Lindquist and J. R. Jenkins. Test-case generation with IOGen. IEEE Software, 5 (1), 72-79, 1988.
8. C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice–preliminary assessment. ICSE '11, 2011.
9. J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. Lecture Notes in Computer Science, vol. 7186, pp 396-411, 2012.
10. M. R. Girgis. An experimental evaluation of a symbolic execution system, Software. Engineering Journal. 7(4), 285-290, 1992.
11. M. R. Girgis, Using symbolic execution and data flow criteria to aid test data selection. The Journal of Software Testing, Verification and Reliability, 3(2), 101-112, 1993.
12. Bertolino, and M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Transactions on Software Engineering, 20 (12), 885-899, 1994.
13. D. F. Yates, N. Malevris. Reducing the effects of infeasible paths in branch testing. ACM SIGSOFT Software Engineering Notes, vol. 14, , pp. 48-54, 1989.
14. Y. Jia, and M. Harman. Higher order mutation testing. Journal of Information and Software Technology, Vol. 51, 10, pp. 1379–1393, 2009.
15. Burnstein. Practical software testing: a process-oriented approach. Springer, 2003.
16. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, vol. 11, 4, pp. 34–41, 1978.
17. R. G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering SE-3, vol. 4, pp. 279–290, 1977.
18. M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. Proceedings of Eighth International Conference on Software Engineering, IEEE Computer Society, pp. 313-319, 1985.
19. M. S. Hecht. Flow analysis of computer programs. Elsevier North Holland, New York, 1977.
20. L. J. White and E. I. Cohen. A domain strategy for computer program testing. IEEE Transactions on Software Engineering, 6(3), 247-257, 1980.
21. C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later" Communications of the ACM, Vol. 56 Issue 2, pp. 82-90, 2013 .