A New Mutant Generation Algorithm based on Path Coverage for Mutant Reduction

Xu, Qin

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
15
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2020-03-24
(URL)
https://doi.org/10.15002/00022719

A New Mutant Generation Algorithm based on Path Coverage for Mutant Reduction

Xu Qin
Graduate School of Computer and Information Sciences
Hosei university
Tokyo, Japan
qin.xu.6d@stu.hosei.ac.jp

***Abstract*—Mutation testing is a fault-based white-box testing technique that can be applied to evaluate the adequacy of a given test suite, but its application is very time-consuming due to the necessity of generating and executing a great number of mutants. How to reduce the cost still remains a challenge for research. In our research, we present a new mutant generation algorithm based on the basic path coverage to reduce mutants. The algorithm is characterized by implementing a basic path segment identification criterion for determining appropriate program points at which faults are inserted and a mutant generation priority criterion for selecting proper mutant operators to make a fault for insertion. We discuss the algorithm by analyzing how the two criteria are realized based on analyzing the control flow graph (CFG) of the program under test and applying effective mutation operators on the appropriate statements in the relevant path segments. We also present an automated mutation testing tool that supports the proposed approach, and a small experiment to evaluate our tool by comparing it with a traditional mutation testing method on six programs. The result of the experiment suggests that using the method of this paper, the high mutation score can be maintained while reducing the number of mutants.**

***Keywords*—Mutation Testing, Path Coverage, Mutant Reduction, Control Flow Graph**

I. Introduction

Mutation testing [1] is a effectively white-box testing technique which can be used to evaluate and improve test suite's adequacy, and predict the possible faults present in our system. As a testing technique, mutation testing can truly reveal various flaws of software. But in industry the mutation testing technique has not been widely applied [2][3]. The main reason is that mutation testing is so time-consuming (a large number of mutants and long execution time).

In recent years, many researches have been carried out in order to apply mutation testing in practical application, including mutant random selection [10], high-order mutant [12], mutant clustering methods [13], selective mutation operators [11], mutant detection optimization, mutant compilation optimization , and parallel execution of mutants.

Although the above mutant reduction methods have been used, there is little work on combining the mutation testing with the conventional path coverage testing. In our research, an algorithm for generating mutants based on the basic path coverage is proposed. We first introduce the criteria for identifying the path segments where faults need to be inserted and the mutant generation priority criteria for producing mutants. We then present a mutant generation algorithm under the constraint of these criteria. The main idea is to insert simple syntactic changes on each basic path of the given program to produce mutants. Combining the traditional path testing coverage with the mutation testing can not only assess the efficiency of the ability of test suite for detecting some possible faults, but also achieve basic path coverage which can effectively improve the effectiveness of given test case set.

It is difficult to do mutation testing without a software tool. So we present an a mutation testing tool that supports the proposed approach, which can generates and executes the mutants automatically. We design three components for our tool: the mutant generator, the mutant viewer, and the test executor. The mutant generator component generates mutants automatically by using the effective mutation operators. It takes Java files as input and a set of mutants as output. The mutant viewer lists the information of our generated mutants. And it also show the original code and the code of each mutant, which can help us to know which part was changed. The test executor run the test case set on generated mutants and shows the test result by analyzing the mutation score of our test case set. All this three components provide GUI for testers to use. We have carried out an experiment on the tool for validating the effectiveness of our proposed approach. From the result we can see that our method can significantly improve the efficiency of mutation testing by reducing mutants but also with high mutation score.

Here lists our contributions:

- Proposing a new mutant generation algorithm by combing the mutation testing and basic path coverage testing for reducing generated mutants.
- Designing and implementing a mutation testing tool to validate our proposed method.

The rest of the paper has the following organization. Section 2 introduces the related work on reducing the mutation testing cost. Section 3 gives some definitions and discuses the implementation process of our proposed mutant generation algorithm based on basic path coverage. Section 4 designs and implements a mutation testing tool to support the proposed approach. Section 5 presents an empirical research to evaluate the efficiency of our proposed method. The conclusion of this paper and the future direction of our research are shown in Section 6.

* Supervisor: Prof. Shaoying Liu

II. RELATED WORK

In recent years, many research have been carried out in order to reduce the cost and improve the efficiency of mutation testing,. In this section, we briefly introduce the research progress of test optimization techniques to reduce the number of mutants.

Acree [10] proposed a method called Mutant Sampling, which randomly select a certain proportion of mutants from all mutants generated for mutation testing. This method can effectively reduce the number of mutant but with lower mutation score.

Hussian [13] proposed a method called the method of Mutant Clustering which classify mutants with similar characteristics, and then randomly select a part of variants from each class for mutation testing. Experiment shows that the clustering method can achieve a good reduction of the number of mutants without affecting the validity of the mutation testing.

Mathur [11] proposed a method which select partial mutation operators applied for mutation testing. This method of generating fewer mutants using a small number of mutation operators is called constraint mutation. Offut et al. further proposed a method of "selective mutation".

Jia and Harman [12] introduced a Higher Order Mutation method. That is, a high-order mutation consists of multiple single-order mutations, and the use of higher-order mutants instead of single-order mutants can effectively reduce the number of mutants.

Although the above mutant reduction technology can reduce the number of mutants, it can not guarantee the path coverage of the program, which affects the sufficiency evaluation ability of the test case set. In this paper, a mutant generation algorithm based on basic path coverage and control flow analysis is used to select the appropriate sentence segment to be inserted into error of the basic path, which reduce the number of mutants and realize the basic path coverage. It not only assess whether the test case set can kill the mutants, but also assess whether the test case set can achieve basic path coverage.

III. OUR PROPOSED MUTANT GENERATION ALGORITHM

A. PRELIMINARY

In order to facilitate the description of the algorithm, we first give the following definitions:

Definition 1 (Immediate predecessor node and successor node).

A immediate predecessor node n_h of a node n_i in is a node that satisfy $P_G(n_i) = n_h|(n_h, n_i) \in E$. ($P_G(n_i)$ means the immediate predecessor node of node n_i; E is a set of directed edges in G.)

A immediate successor node n_j of a node n_i in is a node that satisfy $S_G(n_i) = n_j|(n_i, n_j) \in E$.($S_G(n_i)$ means the immediate successor node of node n_i.)

Definition 2 (Leaf node, sequence node and selection node).

A leaf node n_i is a node that satisfy $S_G(n_i) = \emptyset$ and $\exists! P_G(n_i)$. It means a leaf node only have one input edge and no output edge.

A sequence node n_i is a node that satisfy $\exists! P_G(n_i)$ and $\exists! S_G(n_i)$. It means a sequence node only has one input edge and one output edge.

A selection node n_i is a node containing a condition. It means a selection node have more than one output edge. Note that compound Boolean expressions generate at least two predicate node in control flow graph.

Definition 3 (Sequence path-segment, Unique path-segment).

A sequence path-segment $sp = (n_1, n_2, ...n_n)$is a path segment that the first node is a selection node and the other nodes n_i of sp is either a sequence node or a leaf node.

A unique path-segment is a path-segment that satisfy $ups_i = (N_i, E_i)|E_i \notin \forall ups_j$. It means that any edges of a ups is unique from other unique path-segments.

B. THE FAULT INSERTION PATH SEGMENTS IDENTIFICATION CRITERION AND MUTANT GENERATION PRIORITY CRITERION

In order to generate mutants, we need to find the basic path segments in which faults are to be inserted and then use appropriate operators to generate mutants for the basic path segments we marked above. Here, we introduce a fault insertion basic path segments identification criterion for determining appropriate program segments and a mutant generation priority criterion for selecting proper statements for fault insertion. Also for each rule, a simple example will be given.

Fault insertion path segments identification rule 1 (R1).

If there exists a leaf node n_i in the control flow graph, then trace back to it's immediate predecessor node n_h, if n_h is a sequence node, then continue to trace back to it's immediate predecessor node until we meet a non sequence node n_a, and mark this sequence-path segment $sp = (n_a, ..., n_h, n_i)$ as a path segment to be inserted into fault.(a fault is a simple syntactic change).

The following figure Fig 1 illustrates the application of R1: node 5 is a leaf node and trace back to it's immediate predecessor node 4 and continue to trace back to the non sequence node 2. Then marks the sequence-path segment $sp = (n_2, n_4, n_5)$ as a path segment for fault insertion.

Fault insertion path segments identification rule 2 (R2).

Find all the unique path-segment ups $ups_i = (N_i, E_i)|E_i \notin \forall ups_j$ in each basic paths in the control flow graph CFG and mark this ups as a path segment. If there is a loop structure in the program, the basic path only includes no loop and one loop.

The following figure Fig 2 illustrates the application of the rule R2: we can find all the basic paths, path1:1-7; path2:1-2-3-6-7; path3:1-2-4-6-7; path4:1-2-4-5-6-7 ,and then find the unique path-segment ups1:1-7; ups2:2-3-6; ups3:4-6; ups4:-4-5-6. These unique path-segments are path segments suitable for fault insertion.

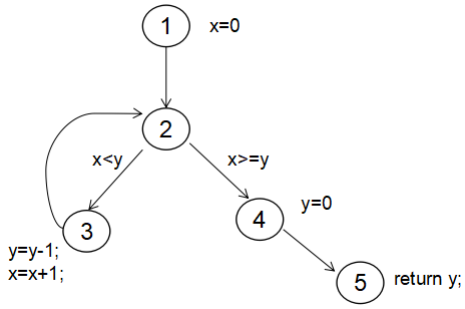

Fig. 1. A example for identification rule1

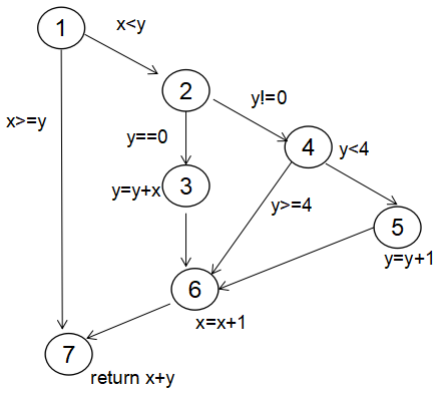

Fig. 2. A example for identification rule2

Mutant generation priority criterion 1 (P1).

For each path-segment of path-segments set $ps = (ps_1, ps_2, ..., ps_n)$, if there is a sequence node n_i at the ps_i, we insert the fault at the statement of n_i. And if $n_i = (s_1, s_2, ..., s_n)$ (s means statement), we insert a fault at the first statement.

As shown in Fig.2: For the path segment ps1:2-3-6, using P1 we find a single node 3, and use the appropriate mutation operator AOM to generate the mutant ,the statement 'y=y+x' of the node3 will be mutated to 'y=y-x' .

Mutant generation priority criterion 2 (P2).

For each path-segment of path-segments set $ps = (ps_1, ps_2, ..., ps_n)$, if there is no sequence node and there is a selection node (predicate operation) n_i at the ps_i, use the appropriate mutation operator to insert the simple syntactic change (fault) at the selection node n_i.

As shown above, for the path segment ps2:4-6, using P2 we find a selection node 4, and generate the mutant 'if($y \leq 4$)' for the node 4 statement 'if($y < 4$)' using the appropriate mutation operator CBM.

Mutant generation priority criterion 3 (P3).

If there is no sequence node and no selection node in the ps_i, inserted fault in the first statement of remaining nodes which are suitable to be inserted fault.

C. THE MUTANT GENERATION ALGORITHM

Applying the fault insertion basic path segments identification criterion for determining appropriate program segments at which faults are inserted and a mutant generation priority criterion for selecting proper statements to make a fault for insertion above, we propose an algorithm whose input is a program and the output is a mutant set.

Algorithm 1 Basic Path Coverage based on Mutant Generation Algorithm

Input:
Original program, P;
Output:
Mutant set, M;
Function mutant-generation(program p) {
Draw CFG for each functions in original program p;
Set FPS=Ø;
for each $CFG_i \in CFG$ **do**
  initialize $FPS_i = \emptyset$;
  if $\exists SPinCFG_i$ **then**
    add sp into FPS_i, $FPS_i \rightarrow FPS_i + sp$;
  end if
  for each $CFG_i inCFG$ **do**
    find all the unique path-segment ups;
    add ups into FPS_i, $FPS_i \rightarrow FPS_i + ups$;
  end for
end for
add FPS_i into FPS, $FPS = (FPS_1, FPS_2, ..., FPS_n)$;
for each ps in EPS **do**
  Set M=Ø;
  if $\exists sequencenode$ **then**
    generate mutant m_i from the sequence node statement s_i for fault insertion;
    add m_i into M, $M \rightarrow M + m_i$;
  else if $\exists selectionnode$ **then**
    generate mutant m_j from the selection node statement s_j for fault insertion;
    add m_j into M, $M \rightarrow M + m_j$;
  else
    generate mutant m_k from the selection node statement s_k for fault insertion;
    add m_k into M, $M \rightarrow M + m_k$;
  end if
end for
Return M;
}

The algorithm is shown above. First, draw the control flow graph CFG of the original program. Then, using the basic path segments identification criterion to determine appropriate program points at which faults are inserted by analyzing the program control flow graph(CFG) and find the appropriate fault insertion path segments in the CFG. Finally, using the mutant generation priority criterion to generate mutants, it

means selecting the appropriate mutation operator to generate mutant at the appropriate statement of the path segments we marked above. The steps of this algorithm are shown as follows:

Step 1. For the original program p, we divide it into some program modules or functions $p = (f_1, f_2, ..., f_n)$ and draw a CFG for each module or function $CFG = (CFG_1, CFG_2, ..., CFG_n)$.

Step 2. Do analysis for CFG_i, find the appropriate fault insertion basic path-segments FPS_i in the CFG.

- First Initialize FPS_i to empty, if the CFG have leaf node, apply the fault insertion basic path segments identification rule 1 (R1) to find a sequence path-segment sp, add sp into FPS_i;
- Then apply the rule2 to find unique path-segments ups on the CFG, add each ups into FPS_i;
- Then we can get a fault insertion path-segment set $FPS_i = (sp_1, ups_1, ups_2, ..., ups_n)$.

Step 3. Repeat step 2 for each CFG_i to get the total fault insertion path-segment set $FPS = (FPS_1, FPS_2, ..., FPS_n)$ for the original program.

Step 4. Do analysis for each path-segment ps_i in the fault insertion path-segment set FPS using the mutant generation priority criterion above and get a mutants set M_i.

- apply P1, generate mutant for a sequence node statement using appropriate mutation operator. For example, MAO, IO, VMC;
- apply P2, generate mutant for a predicate node statement;
- apply P3, generate mutant for the first statement of remaining node that can be inserted into fault.

Step 5. For each path-segment ps_i in the fault insertion path-segments set FPS, Repeat the above step 4) to get the mutants set$M = M_1, M_2, ..., M_n$ of the original program P.

IV. TOOL IMPLEMENTATION

Our proposed algorithm is aiming at reducing the cost of mutation testing by generating less but effectively mutants.In order to validate the efficiency and effectiveness (accuracy of mutation score), we implement an automated mutation testing tool to support our algorithm. It takes the program and test case set as input, does the mutation testing automatically, and finally produces a analysis report to show the test result.

The main functions of this automated mutation testing tool are using effective mutation operators to generate mutants (mutant generation), executing the given test case set on the mutants (mutant execution), showing the result and report of mutation testing (result analysis). Figure 4 shows the overall structure of our mutation testing tool. It is composed of 3 components.

The mutant generator generates mutants by using the effective mutation operators. It generate mutants for selected java files. The GUI for the mutant generator can help us to choose which project and which files under test. The mutant viewer component lists the detailed information for each generated mutant including operatorType, lineNumber, description and so on. And it also shows the code of original program and mutant which help us to know which statement of program under test is mutated and design test cases to kill the generated mutants. The test executor runs the test case set on generated mutants and reports the testing result by computing the mutation score of given test case set.

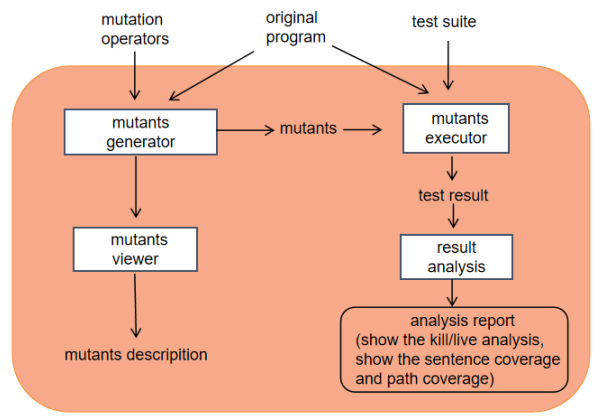

Fig. 3. The overall structure of the automated mutation testing tool.

A. MUTANTS GENERATOR

According to the above design, we implement the tool using Java Swing and the user interface are shown as follows. Figure 5 shows how the Mutants Generator works. We can select the project and a set of Java files under test to create mutants, view the description of applied mutation operators, and press the "Generate" button to prompt the tool to generate mutants. The Mutants Viewer panel will show the information for each mutant after generation.

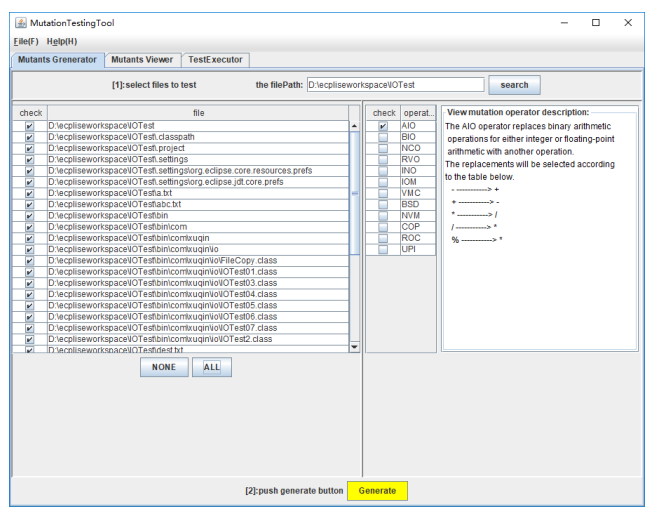

Fig. 4. The Mutants Generator GUI.

B. MUTANTS VIEWER

The Mutants Viewer panel in Figure 6 lists all the generated mutants and shows some detailed description of each mutant.

It help us to analyze mutants by displaying the information of each mutant and which statement of given program is changed by the mutant. It is divided into two parts. The upper part is a mutants list which shows a brief descriptions of the each mutant including a operatorType, className, description and lineNumber. The lower part shows the original code and the mutant. By choosing a mutant in the mutants list of the upper part, the lower part will show the original java file and the mutant, which helps testers to know which statement is mutated, design test cases to kill mutants which are difficult to kill.

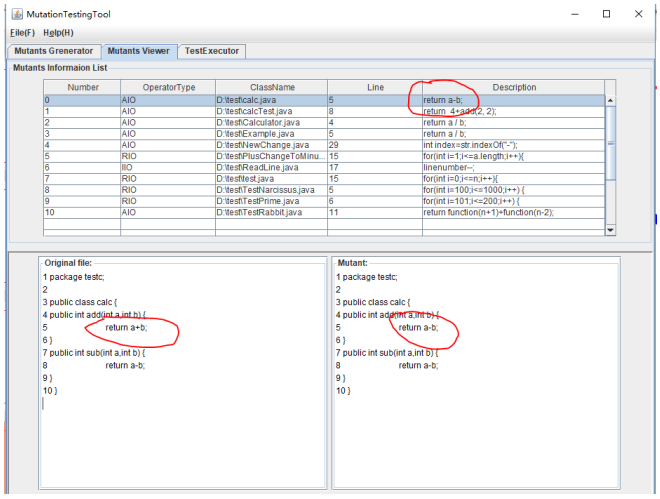

Fig. 5. The Mutants Viewer GUI.

C. TEST EXECUTOR

Figure 7 shows the GUI of the Test Executor panel. Lower left part shows the number () of mutants generated by different operator type. The lower right part shows the results of mutation testing and the number of live mutants and dead mutants. Also, the tester can export the result into a HTML file, which shows test results more clearly and can be saved as important test document. The testing report is shown in Figure 8.

V. EXPERIMENT FOR OUR METHOD

In order to assess the effectiveness of our proposed method, 6 benchmark programs were selected as the tested programs, all of which were written in JAVA language. Feasibility and effectiveness were assessed using empirical and comparative studies.

A. RESEARCH QUESTIONS

Our experiment described here mainly focuses on the following Research Questions (RQs):

RQ 1. Can our algorithm proposed in this paper effectively reduce the mutant's number? (by calculating the reduction rate of mutants)

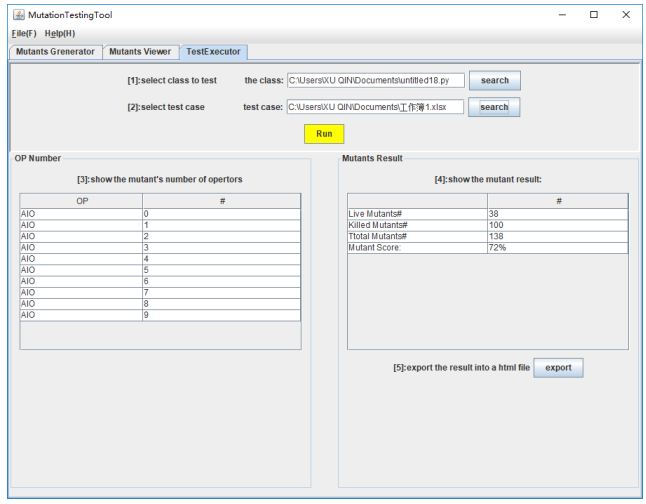

Fig. 6. The Test Executor GUI.

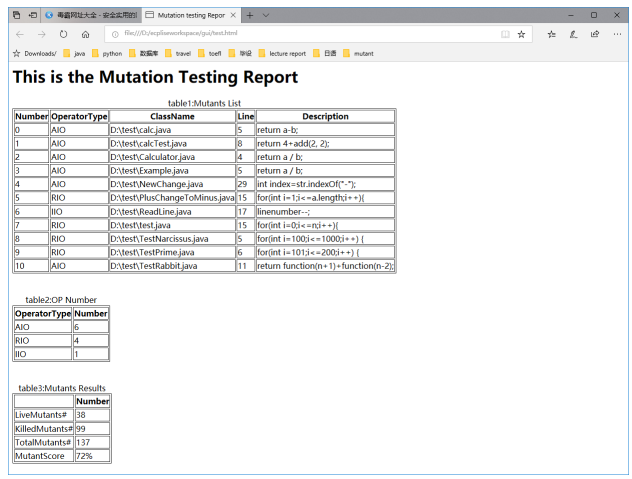

Fig. 7. The testing report in a HTML file.

We proposed a mutant reduction rate to assess the ability of the mutant reduction. The mutant reduction rate is:

$$MRR = \frac{M_t - M_p}{M_t} \tag{1}$$

M_t: mutant's number generated in traditional method, M_p: mutant's number generated by the proposed method. The MRR shows that the higher the reduction rate is, the better outcomes the effectiveness of this method in reducing mutants.

RQ 2. Can a test case set that kills mutants generated by the proposed algorithm be able to kill mutants of traditional methods ?

Here,we use the mutation score as evaluation index:

$$MS = \frac{M_K}{M_a - M_e} * 100 \tag{2}$$

M_k: the number () of killed mutants, M_a: the number () of all generated mutants, M_e: the number of () all equivalent mutants.

The test case set used in this paper's experimental evaluation is constructed using traditional test case generation algorithms,

such as boundary value analysis, statement coverage, and branch coverage.

B. EXPERIMENT RESULT ANALYSIS

As shown below, we clearly see that mutant's number in this method is much smaller than traditional methods.

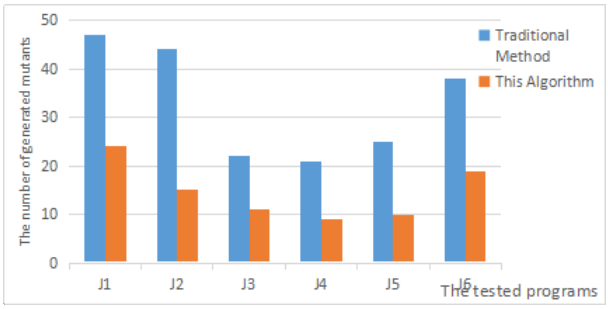

Fig. 8. The comparison of generated mutants.

As shown in Table 1, J2 obtained the largest reduction rate of 65.9% in 6 programs, J1 obtained the minimum reduction rate of 48.9%, all programs reached a high reduction rate, and we can see the average reduction rate of six programs is 55.3%, which suggests that our proposed method has a high efficiency in reducing generated mutant's number.

TABLE I
THE REDUCTION RATE OF EACH EXPERIMENTAL SUBJECT.

ID	Number of mutants in traditional method	Number of mutants in proposed method	Reduction Rate
J1	47	24	48.9%
J2	44	15	65.9%
J3	22	11	50%
J4	21	9	57.1%
J5	25	10	60%
J6	38	19	50%

TABLE II
THE MUTATION SCORE OF EACH EXPERIMENTAL SUBJECTS.

ID	Mutants #	Killed Mutants#	Mutation Score%
J1	47	43	91.4%
J2	44	39	88.6%
J3	22	20	90.9%
J4	21	19	90.5%
J5	25	24	96%
J6	38	35	92.1%

The table 2 shows the detection results of the test case set that can detect the mutants generated by the proposed method on the traditional mutants set. The mutation score is used as the evaluation index. It can be seen that for all the tested programs, the average mutation scores exceed 91.5% and only a small number of mutants(55.3%) are used. So using the method of this paper, the high mutation score can be maintained while generating less mutants.

VI. CONCLUSION

In this paper, a mutant generation method based on basic path coverage is proposed for reducing generated mutant's number in mutation testing. Different from the previous methods, by analyzing the control flow structure and basic path of the source program, an identification of path segments suitable for fault insertion and a priority criteria for generating mutants are proposed. By using these criteria to select some appropriate statements for mutation operation, the mutants needed to kill is reduced and the coverage of the basic path is achieved, which improves the effectiveness of the mutation testing. In order to evaluate the efficiency (mutant's number) and effectiveness (accuracy of mutation score) of our proposed method, we implement an automated mutation testing tool to support our algorithm. Our method was applied to 6 tested programs, and the results showed that using the method of this paper, the high mutation score can be maintained while reducing the number of mutants.

This work has opened up a research direction of mutation test optimization technology. The next steps include using more efficient mutation operators to generate mutants, using a larger industrial application sample program to evaluate the effectiveness of the method, and exploring other more efficient mutant reduction techniques.

REFERENCES

[1] Jia, Y. & Harman, M. . (2011). An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 37(5), 649-678.

[2] Offutt, A. J. , & Untch, R. H. . (2001). Mutation 2000: Uniting the Orthogonal. Mutation Testing for the New Century. Kluwer Academic Publishers.

[3] Just, René, Ernst M D , Fraser G . Efficient mutation analysis by propagating and partitioning infected execution states.[C]// 2014.

[4] Siami Namin A , Andrews J H , Murdoch D J . [ACM Press the 13th international conference - Leipzig, Germany (2008.05.10-2008.05.18)] Proceedings of the 13th international conference on Software engineering, - ICSE Ö8 - Sufficient mutation operators for measuring test effectiveness[J]. 2008:351.

[5] Allen, F. E. . (1970). Control flow analysis. Acm Sigplan Notices, 5(7), 1-19.

[6] Zapata, F. , Akundi, A. , Pineda, R. , & Smith, E. . (2013). Basis path analysis for testing complex system of systems. Procedia Computer Science, 20(Complete), 256-261.

[7] Papadakis M , Malevris N . Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing[J]. Software Quality Journal, 2011, 19(4):691-723.

[8] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[9] Fraser G , Zeller A . Mutation-Driven Generation of Unit Tests and Oracles[J]. IEEE Transactions on Software Engineering, 2012, 38(2):278-292.

[10] A.T. Acree.On Mutation. PhD dissertation, Georgia Inst. of Technology, 1980.

[11] Mathur A P . Performance, effectiveness, and reliability issues in software testing[C]// International Computer Software & Applications Conference. IEEE, 1991.

[12] Jia Y , Harman M . Constructing Subtle Faults Using Higher Order Mutation Testing[C]// Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on. IEEE, 2008.

[13] Hussain S.Mutation Clustering[Ph.D. dissertation]. King's College, london,UK,2008.