Mutant Reduction Based on Dominance Relation for Weak Mutation Testing

Dunwei Gong[a,b,*], Gongjie Zhang[c,e], Xiangjuan Yao[d], Fanlin Meng[f]

[a] School of Information and Electrical Engineering, China University of Mining and Technology, Xuzhou, Jiangsu, 221116, China.
[b] School of Electrical Engineering and Information Engineering, LanZhou University of Technology, Lanzhou, Gansu, 730050, China.
[c] School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu, 221116, China.
[d] School of Science, China University of Mining and Technology, Xuzhou, Jiangsu, 221116, China.
[e] School of Computer Science and Technology, Jiangsu Normal University, Xuzhou, Jiangsu, 221116, China.
[f] Department of Aeronautical and Automotive Engineering, Loughborough University, Loughborough, LE11 3TU, UK.
[*] Corresponding author, email address: dwgong@vip.163.com.

Abstract

Context: As a fault-based testing technique, mutation testing is effective at evaluating the quality of existing test suites. However, a large number of mutants result in the high computational cost in mutation testing. As a result, mutant reduction is of great importance to improve the efficiency of mutation testing.
Objective: We aim to reduce mutants for weak mutation testing based on the dominance relation between mutant branches.
Method: In our method, a new program is formed by inserting mutant branches into the original program. By analyzing the dominance relation between mutant branches in the new program, the non-dominated one is obtained, and the mutant corresponding to the non-dominated mutant branch is the mutant after reduction.
Results: The proposed method is applied to test ten benchmark programs and six classes from open-source projects. The experimental results show that our method reduces over 80% mutants on average, which greatly improves the efficiency of mutation testing.
Conclusion: We conclude that dominance relation between mutant branches is very important and useful in reducing mutants for mutation testing.

Keywords: software testing, weak mutation testing, mutant, reduction, dominance relation

1. Introduction

Software testing, which is used to seek existing defects or faults in software before it is released to the market, is an important way to improve software quality. Mutation testing is commonly used to evaluate the quality of existing test suites to guide testers how they might be able to improve them [1]. Compared with other structural coverage criteria, test suites that are mutation adequate can reveal more faults [2]. It is noticeable that mutation testing has attracted widespread attention from researchers and developers in both academia and industry.

Mutation testing is a fault-based technique [3, 4], and the related concepts are given as follows. By making a simple syntactic change to the original program, a mutant is generated. A rule used to perform the syntactic changes is called a mutation operator. If a test datum can distinguish the outputs between a mutant and its original program, the mutant is said to be *killed*. A mutant is equivalent, if it cannot be killed by any test datum. Generally, the adequacy of mutation testing named *mutation score* is defined as the ratio of the number of killed mutants to the total number of non-equivalent mutants.

In order to optimize the execution of the traditional mutation testing, Howden first proposed weak mutation testing [5]. Instead of checking a mutant after executing the whole program, weak mutation testing checks a mutant immediately after executing the mutated statement.

In mutation testing, mutants are employed to reflect possible real faults in software under test [6–8]. Many lines of code (LOCs), complicated statements, and a variety of data types [9] in software greatly increase the number of mutants. It results in the high computational cost in mutation testing, therefore the mutant reduction is of great

importance and necessity. Although there have been several techniques for mutant reduction [10–16], their efficiency needs to be further improved.

Just et al. focused on the COR and ROR mutation operators to identify redundant mutants [13]. Kaminski et al. sought a subset of relational operators that subsumes the others to reduce mutants [14]. Focusing only on a subset of mutation operators opens new research directions [13, 14]. Papadakis and Malevris transformed the problem of killing mutants into the problem of covering mutant branches in the new program, and generated test data by conventional approaches [17]. Although it is relatively efficient, a large number of mutants without reduction, will inevitably add high complexity to the new program.

In the previous work on dominance analysis, Marre and Bertolino employed the subsumption relation between entities in a ddgraph (a simplified control flow graph) to seek the minimal set of entities named the spanning set, so as to reduce the number of entities needed to cover [18]. In addition, Ghiduk and Girgis identified the non-dominated nodes in a control flow graph (CFG) by analyzing the dominance relation between nodes [19]. Both the above methods are performed among the original entities (nodes) for structural coverage testing. Different from the above work, we analyze the dominance relation between mutant branches, which are instrumented branches transformed from mutants based on the method presented by Papadakis and Malevris for weak mutation testing, with the aim to reduce the number of mutants, and to improve the efficiency of testing.

Considering all the traditional (method level) mutation operators, we first construct mutant branches based on the statements before and after mutation, and form the new program by fusing all mutant branches into the original program using the method proposed by Papadakis and Malevris [17]. Then, we identify redundant mutants according to the dominated mutant branches after manual analysis with the aid of the dominance relation graph. Mutants associated with the non-dominated ones will remain. The test data that cover the non-dominated mutant branches can also cover all the mutant branches, i.e., kill all the mutants before reduction in weak mutation testing.

The basic idea of defining the dominance relation between mutant branches and applying the dominance relation to reduce mutants was initially reported, with examples on several small programs, at the 2nd Chinese Search Based Software Engineering (CSBSE'2013) workshop [20]. Given the fact that the 2-page abstract is preliminary, we have extended the idea in the following four new directions:

(1) defining four concepts, mutant branch, dominance relation, non-dominated branch, and dominance relation graph;

(2) presenting two theorems on how to form the non-dominated mutant branch set and identify the non-dominated mutants;

(3) providing an example throughout the whole paper to intuitively demonstrate the above work;

(4) evaluating the proposed method by applying it to ten benchmark programs and six classes from open-source projects with various sizes and complexities.

The main contributions of this paper are as follows:

• A method of reducing mutants is proposed for weak mutation testing, which is conducted by analyzing the dominance relation between mutant branches in the new program.

• Four definitions of identifying the dominance relation between mutant branches are provided, and the dominance relation graph is given to describe all the dominance relations in the new program.

• Two theorems of determining the non-dominated mutant branches are given, so as to reduce redundant mutants.

• The proposed method is applied to ten benchmark programs and six classes from open-source projects, and the experimental results suggest that our method reduces over 80% mutants.

2. Related Work

Reducing mutants is of effectiveness to save computational cost for mutation testing. Weak mutation testing is a technique in view of saving execution time. Additionally, there are correlations among statements in a program, and correlation analysis is helpful to mutation testing. This section will review the related work from the above aspects.

2.1. Mutant Reduction

For software under test, a large number of mutants cause high cost in mutation testing, which can be solved by mutant reduction. Mutant sampling proposed by Acree and Budd randomly selects a specific percentage of mutants to

execute testing [21, 22]. Mathur and Wong investigated the influence of the sampling rate on the mutation adequacy [23]. They conducted a series of experiments by changing the rate from 0.1 to 0.4 in the step of 0.05, and the experimental results suggest that the mutation score decreases as the sampling rate reduces. Unlike random sampling, Hussain extracted a set of representative mutants to test after clustering, and his method maintains a high mutation score [24].

To reduce mutants, Mathur suggested that two mutation operators (i.e., Array reference for Scalar variable Replacement (ASR) and Scalar Variable Replacement (SVR)), which will generate around 30% to 40% of the total mutants, should be omitted, and proposed selective mutation testing [25]. By Extending Mathur's idea, Offutt et al. performed 2-selective experiments (omitting ASR and SVR) and achieved 24% mutant reduction with 99.99% mutation score. Further in 4-selective and 6-selective experiments, they got much higher reduction rates but lower mutation scores [26]. By analyzing the distribution of 22 mutation operators in 28 programs, Offutt et al. obtained a high mutation score with only 5 operators [27]. Different from the prior selective mutation that reduces mutants with acceptable loss in the test adequacy, Mresa and Bottaci compared mutation operators in terms of both score and cost to seek the most efficient mutation operators [28]. By manual analysis, Yao et al. revealed a highly uneven distribution of equivalent mutants and stubborn mutants (those that remain undetected by a test suite with high quality, yet are non-equivalent) [29]. Their work is beneficial to designing mutation tools, and reduces the human effort involved in mutation testing.

The above techniques, mutant sampling and selective mutation, are from First Order Mutants (FOMs) perspective. In view of the fact that High Order Mutants (HOMs, more than one syntactic change in a program) not only represent complex faults in practical software, but also reduce the number of mutants [30], studies on HOMs have arisen in recent years. Jia and Harman utilized meta-heuristic search algorithms to generate semantic HOMs which are hard to kill, and reduced mutants greatly [4]. Langdon et al. combined FOMs with close semantic relations to form HOMs [31]. Second-order mutants (SOMs, two syntactic changes in a program) proposed by Polo et al. get 50% cost saving [32]. Kintis et al. focused on control relations among nodes in the CFG of a program, and presented three strategies for combining SOMs [33]. Papadakis and Malevris conducted an empirical study for the first and the second order mutation testing strategies, and found that the first order mutation testing strategies are generally more effective than the second order ones, and the latter drastically reduce equivalent mutants, thus forming a valid cost effective alternative to mutation testing [34]. However, the growing order of HOMs results in significant cost increase in generating HOMs, which further illustrates the necessity of reducing mutants.

2.2. Weak Mutation Testing

In strong mutation testing, a mutant is killed when its output differs from that of the original program. DeMillo and Offutt summarized the conditions for killing a mutant as *reachability*, *necessity*, and *sufficiency* [35]. Reachability claims that a test datum must execute the mutated statement in the mutant, while necessity requires that an unexpected state must occur after executing the mutated statement. Finally, sufficiency means that the unexpected state must be propagated up to the output of the mutant. For software under test, running code after the mutated statement is usually time-consuming.

Weak mutation testing proposed by Howden [5] only satisfies reachability and necessity. Experiments from DeMillo and Offutt suggest that test data that satisfy necessity can largely satisfy sufficiency [36]. Moreover, Horgan and Mathur's analysis shows that on certain circumstances, test data generated in weak mutation testing are as effective as those in strong mutation testing [37]. Instead of executing the whole program, weak mutation testing greatly saves the execution time. Offutt and Lee's experiments report that weak mutation testing saves about 50% test cost, and in most cases, weak mutation testing is an effective alternative to strong mutation testing [38, 39].

To further improve weak mutation testing, Papadakis and Malevris combined the statements before and after mutation to construct a mutant branch, and formed a new program by inserting all the mutant branches into appropriate positions in the original program. Test data that cover mutant branches also kill the corresponding mutants in weak mutation testing. Their experiments indicate that without much loss in the mutation score, the method further saves execution time [17]. However, a large number of mutant branches in the new program dramatically increases the complexity. Moreover, Papadakis and Malevris generated mutation-based test data by covering the selected paths [40]. Similarly, without reduction, a lot of mutants result in a large search space, which severely increases cost in generating test data.

2.3. Testability Transformation and Mutant Reduction

The transformation method proposed by Papadakis and Malevris saves the cost in executing all mutants one by one. However, numerous mutants are inserted into the program under test, which greatly increase the complexity of the new program. Reducing mutants can further simplify the transformed problem.

Based on the dominance and implication relations in a ddgraph, Bertolino and Marre sought a set of unconstrained arcs, and generated the minimal path set which covers all the branches in a program. By transforming branch coverage into path coverage, testing is highly simplified [41]. Using the ddgraph, Marre and Bertolino proposed the concept of spanning set which is the minimal set under a specific coverage criterion, and test data that cover the spanning set also cover the original set. The experimental results suggest that their approach not only saves testing cost, but also reduces the number of generated test data [18].

Previous research shows that test strategies based on the correlations among statements are of effectiveness. In structural testing, Ghiduk and Girgis reduced the statements needed to cover by only covering the non-dominated ones [19]. Gong and Yao identified infeasible paths according to the correlations among branches to save valuable testing resources [42]. Correlations are also helpful to mutation testing. Shan et al. analyzed the correlations among mutated statements at the same position, and combined a small amount of compound mutants. Their method gets a high reduction rate [43]. Kintis et al. generated SOMs based on the control relations among nodes in the CFG of a program [33]. Their research demonstrates that there exists subsumption between mutants.

Ammann et al. determined the minimal set of mutants by dynamic subsumption analysis, which is performed by executing mutants against test data [44]. Kurtz et al. defined the following three types of subsumption relations, i.e., true subsumption, dynamic subsumption and static subsumption, and built a graph model to represent the subsumption relation between mutants [45]. In addition, they indicated that the true subsumption is not computable, the dynamic subsumption which is an approximation to the true one can be detected by a specific test set, and the static subsumption which approaches the true one can be manually or automatically analyzed. Later, Kurtz et al. developed a method of building the static mutant subsumption graph (SMSG) by the symbolic execution, so as to generate test data to kill mutants. Further, they refined the SMSG to form a statically-derived dynamic mutant subsumption graph (SDMSG) by generating test data in a feedback mechanism [46].

The dominance relation graph used in our paper shares several similar features with MSG. For the dominance relation graph and MSG, each node represents one or more killable mutants, and each edge between nodes indicates the dominance relation between mutants. MSG suggests that describing the dominance relations with a graph is feasible. Nonetheless, the significant difference between the dominance relation graph and MSG is that each node in the dominance relation graph represents only one mutant (mutant branch), whereas each node in MSG refers to one or more mutants that are indistinguishable from each other.

3. The Proposed Method of Reducing Mutants

This section describes the method of reducing mutants by the dominance relation. In this method, we first construct mutant branches based on the statements before and after mutation, and a new program is formed by fusing all the mutant branches into the original program. Then, we analyze the dominance relation between mutant branches in the new program. Finally, we obtain the non-dominated mutant branches which correspond to the mutants after reduction.

3.1. Constructing Mutant Branches and Forming the New Program

Constructing mutant branches and forming the new program is the first step of our approach, which transforms the problem of killing mutants into that of covering mutant branches. The detailed process of the transformation method proposed by Papadakis and Malevris is provided as follows.

The original program is denoted as P. Suppose that the mutant position is statement s, and a mutated statement of s is s'. A mutant of P, denoted as m, is generated by substituting s' for s in P. In weak mutation testing, if a test datum kills m, it must reach s' and cause a different state after executing s', i.e., s != s'. Taking "s != s'" as the predicate, and the marked statement(s), which reflect that m is killed, as the true branch, denoted as b, a conditional statement is constructed. Then the problem of killing m is transformed into that of covering the true part of b. In view of the one-to-one mapping relation between b and s', or equivalently between b and m, b is called a mutant branch.

Definition 1 Mutant Branch

Table 1: The construction process of mutant branches.

Mutation operator	Description	Original statement (s)	Mutated statement (s')	Mutant branch (b)
AORB	Basic Arithmetic Operator Replacement	$a + b$	$a * b$	$if((a + b) != (a * b))$
AORS	Short-cut Arithmetic Operator Replacement	$a + + + b$	$a - - + b$	$if((a + 1 + b) != (a - 1 + b))$
AOIU	Insert basic Unary Arithmetic Operators	$a + b$	$-a + b$	$if((a + b) != (-a + b))$
AOIS	Short-cut Arithmetic Operator Insertion	$a + b$	$+ + a + b$	$if((a + b) != (a + 1 + b))$
AODU	Delete Basic Unary Arithmetic Operators	$-a + b$	$a + b$	$if((-a + b) != (a + b))$
AODS	Short-cut Arithmetic Operator Delete	$a + + + b$	$a + b$	$if((a + 1 + b) != (a + b))$
ROR	Relational Operator Replacement	$a > b$	$a <= b$	$if((a > b) != (a <= b))$
COR	Conditional Operator Replacement	$a > b \| \| c < d$	$a > b \&\& c < d$	$if((a > b \| \| c < d) != (a > b \&\& c < d))$
COD	Conditional Operator Deletion	$!(a > b)$	$a > b$	$if(!(a > b) != (a > b))$
COI	Conditional Operator Insertion	$a > b$	$!(a > b)$	$if((a > b) != !(a > b))$
SOR	Shift Operator Replacement	$a >> b$	$a >>> b$	$if((a >> b) != (a >>> b))$
LOR	Logical Operator Replacement	$a \mid b$	$a \& b$	$if((a \mid b) != (a \& b))$
LOI	Logical Operator Insertion	$a + b$	$\sim a + b$	$if((a + b) != (\sim a + b))$
LOD	Logical Operator Deletion	$\sim a + b$	$a + b$	$if((\sim a + b) != (a + b))$
ASRS	Assignment Operator Replacement	$a+ = b$	$a* = b$	$if((a + b) != (a * b))$

For a statement s in P and its mutated one s', the corresponding mutant branch b is composed of a conditional expression in the form of "$s! = s'$" and a true branch which marks that an input satisfies the conditional expression.

When forming a mutant branch, only the conditional expression and its true part are of necessity. The reason lies that if the true part is covered, it indicates that the corresponding mutant is killed in weak mutation testing. From this point of view, we usually call the conditional expression and its true part mutant branch.

We combine all the mutant branches by the mutated statements and the original ones after performing all the mutation operators on P. The combined mutant branches are denoted as $b_1, b_2, ..., b_N$, and their corresponding mutants as $m_1, m_2, ..., m_N$, where N is the number of mutant branches (mutants). By fusing all the mutant branches into the original program in order, a new program, denoted as P', is formed.

Table 1 lists the construction process of mutant branches by performing traditional mutation operators of MuClipse. Here, MuClipse is a plug-in for Eclipse from MuJava (a widely used mutation testing tool in various research) [47], which can automatically generate and execute mutants [47, 48]. In this table, the original statement s, a mutated statement s' generated by performing a mutation operator on s, and the mutant branch b constructed by combining s and s' are provided. For example, a mutated statement "$a * b$" is generated after performing AORB on "$a + b$", and its corresponding mutant branch, "$if((a + b)! = (a * b))$", is constructed by combining "$a + b$" and "$a * b$".

Several mutation operators in Table 1 may affect the execution result P'. For example, performing AOIS on variable "a" in "$a + b$" will generate "$+ + a$", "$- - a$", "$a + +$" and "$a - -$". In our study, "$+ + a$" and "$- - a$" are directly replaced with "$(a + 1)$" and "$(a - 1)$", respectively. For "$a + +$" and "$a - -$", since they affect the consequent code in P', we replace them with"$(a + 1)$" and "$(a - 1)$", respectively.

The program, Triangle, is a familiar example in software testing [2, 4, 30, 31], and Figure 1 (a) is its Java version. Figure 1 shows the process of forming P' where (a) is the code of P (Triangle); (b) represents the CFG of code from lines 7 to 9 in the dashed box of (a); (c) means the mutant branches constructed after performing ROR on "$if(a == b)$" and AORB on "$trian = trian + 1$", respectively; and (d) is the CFG after inserting mutant branches into P. The dashed boxes in Figure 1 (d) represent the inserted mutant branches. Although each has two branches, i.e., the true and the false branches, we only consider the true one. Further, the statement(s) in the true branch indicates that its true part is covered, i.e., there is an unexpected state in weak mutation testing.

After Forming the new program, P', the problem of killing N mutants of P is transformed into the problem of covering N mutant branches in P'. However, a lot of mutant branches in P' greatly increase the complexity in solving the transformed problem. We use the program in Figure 1 (a) to illustrate the increased structural complexity in P'.

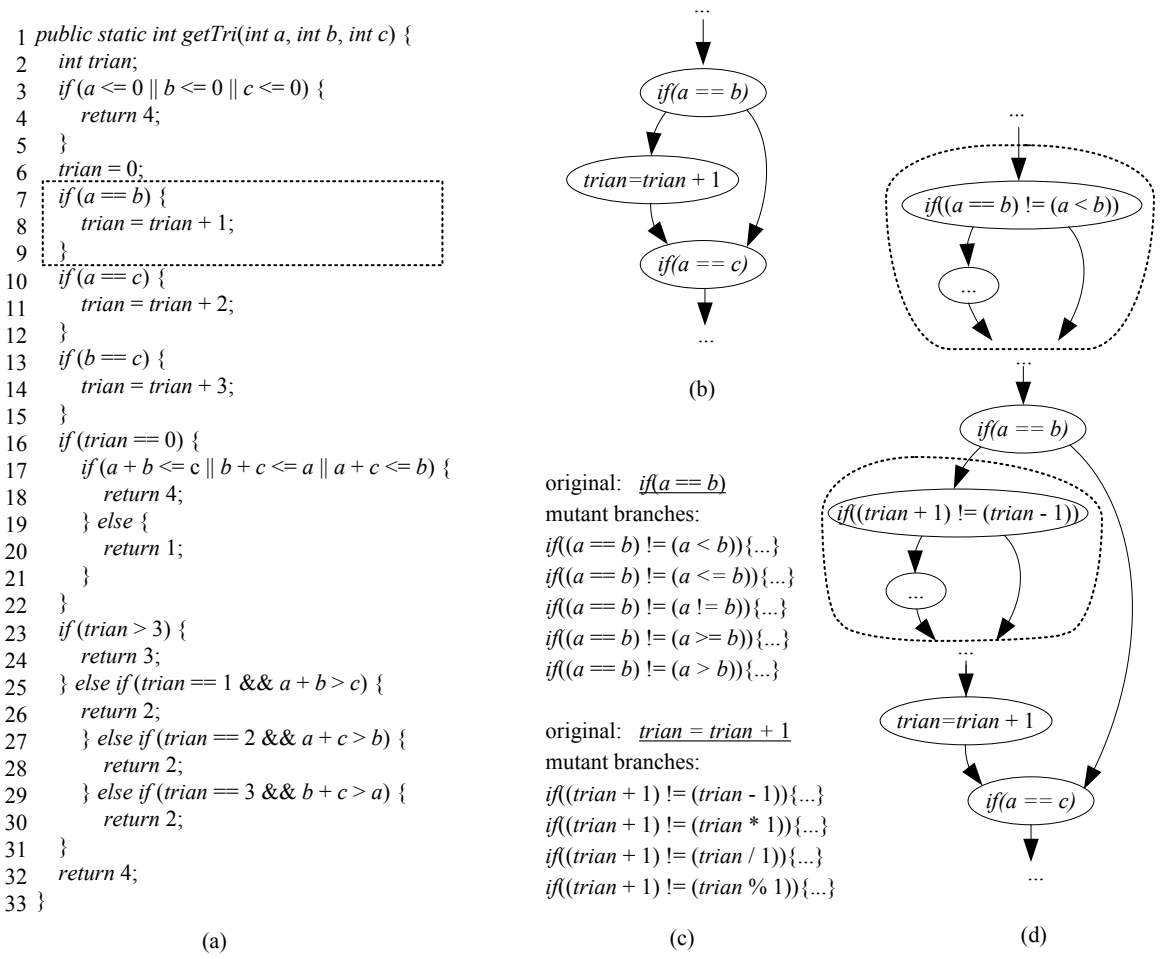

Figure 1: The process of forming a new program. (a) triangle.java translated from [2, 4, 30, 31]. (b) the CFG of lines 7 to 9 in (a); (c) the constructed mutant branches by applying ROR to statement "$if(a == b)$" and AORB to "$trian = trian + 1$", respectively; (d) the CFG after inserting mutant branches.

Table 3: The predicates of the constructed mutant branches.

Mutation operator	Predicate	
AOIS	1. $if((a == b)\ != (++a == b))$;	2. $if((a == b)\ != (- - a == b))$;
	3. $if((a == b)\ != (a == ++b))$;	4. $if((a == b)\ != (a == - - b))$;
	5. $if((trian + 1)\ != (++trian + 1))$;	6. $if((trian + 1)\ != (- - trian + 1)$;
	21. $if((a == b)\ != (a++ == b))$;	22. $if((a == b)\ != (a - - == b))$;
	23. $if((a == b)\ != (a == b++))$;	24. $if((a == b)\ != (a == b - -))$;
	25. $if((trian + 1)\ != (trian++ + 1))$;	26. $if((trian + 1)\ != (trian - - + 1))$.
AOIU	7. $if((trian + 1)\ != (-trian + 1))$.	
AORB	8. $if((train + 1)\ != (trian - 1))$;	9. $if((trian + 1)\ != (trian * 1))$;
	10. $if((trian + 1)\ != (trian / 1))$;	11. $if((trian + 1)\ != (trian \% 1))$.
COI	12. $if((a == b)\ != (!(a == b)))$.	
LOI	13. $if((a == b)\ != (\sim a == b))$;	14. $if((a == b)\ != (a == \sim b))$;
	15. $if((trian + 1)\ != (\sim trian + 1))$.	
ROR	16. $if((a == b)\ != (a < b))$;	17. $if((a == b)\ != (a <= b))$;
	18. $if((a == b)\ != (a\ != b))$;	19. $if((a == b)\ != (a >= b))$;
	20. $if((a == b)\ != (a > b))$.	

All the 15 traditional operators of MuClipse are selected to mutate the program, and 78 mutants are generated from lines 6 to 15 in Figure 1 (a). The mutants are listed in Table 2, where "others" represents the other nine mutation operators which generate 0 mutants. For "$if(a == b)\{trian = trian + 1;\}$" (from lines 7 to 9 in Figure 1 (a)), 26 mutants are generated, where 16 of them are generated from "$if(a == b)$" (line 7) and 10 from "$trian = trian + 1$" (line 8). The number of constructed mutant branches is also 26, and their predicates are listed in Table 3. Figure 2 shows the simplified CFG of the new program by integrating these mutant branches into the original program (lines 7 to 9), where each number shown in Figure 2 corresponds to a predicate in Table 3.

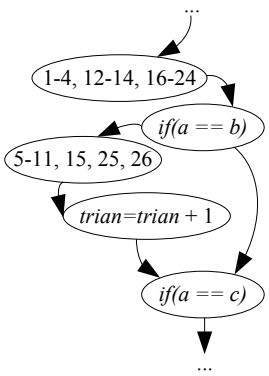

Figure 2: The CFG of the new program.

Mutation operator	Mutants
AOIS	33
AOIU	3
AORB	12
COI	3
LOI	9
ROR	15
others	0
Total	78

Table 2: The mutation operators and the number of mutants they generate.

According to Table 1, the mutant branches generated by AOIS in Table 3 need further transformations. For a mutated statement, if there is no variable reference in the consequent code, or the effect cannot be propagated, its mutant branch is infeasible, and the corresponding mutant is said to be equivalent. As a result, statements like "$trian = trian + + + 1$" and "$trian = trian - - + 1$" (corresponding to predicate 25 or 26 in Table 3) are infeasible. Further predicate 7 in Table 3 is also infeasible, due to that the initial value of "$trian$" is 0.

After 78 mutant branches from 10 lines of code (from lines 6 to 15 in Figure 1 (a)) are added to P', the number

of branches is increased by 26 times (the original program includes three branches from lines 6 to 15, which is shown in Figure 1 (a)). Since each mutant branch includes at least three lines of code, the number of lines in P' is at least increased 23 times ($78 \times 3 \div 10$), which suggests that it is of necessity to reduce mutant branches in P'.

To simplify P', it is critical to seek the dominance relation between mutant branches. The next subsection will analyze the dominance relation in P'.

3.2. Determining the Dominance Relation between Mutant Branches

Although the fusion of mutant branches increases the structural complexity, the branches which include the original and the mutant ones in P', are not completely independent. For example, in Table 3 and Figure 2, if mutant branch 17 is executed, 16 and 18 must be executed. The reason is given as follows. First, the above three branches have the same reachability condition. Second, the conditional expression of 17 can be simplified as "$a < b$", while those of 16 and 18 can be simplified as "$a == b \parallel a < b$" and "$a == b \parallel a < b \parallel a > b$". Since "$a < b$" is the sufficient condition of both "$a == b \parallel a < b$" and "$a == b \parallel a < b \parallel a > b$", any test datum that executes the true branch of 17 must also execute those of 16 and 18.

Additionally, if (the true part of) mutant branch 5 is executed, mutant branches 6, 9 and 10 must also be executed. The above observations lead to the conclusion that (1) the correlation exists between mutant branches in P', and (2) such correlation is useful in identifying redundant mutant branch. Note that the above correlation is called the dominance relation.

Definition 2 Dominance Relation

Consider two mutant branches b_i and b_j in P', for any test datum, if b_i is executed, b_j must be executed, b_i is said to dominate b_j, denoted as $b_i \succ b_j$. In this case, b_i is called the dominating branch, and b_j the dominated branch.

From Definition 2, the dominance relation exists only between feasible mutant branches. When defining the dominance relation, we highlight that a dominating mutant branch must be feasible. This is because if a mutant branch is infeasible, it will neither dominate any other mutant branch, nor be dominated by any other mutant branch. As a result, we can conclude that the definition neither loses any dominated mutant, nor reduces the mutation score. The dominance relation has the properties of reflexivity and transitivity described as follows:

(1) $b_i \succ b_i$;

(2) for three branches b_i, b_j, and b_k in P', if $b_i \succ b_j$ and $b_j \succ b_k$, one has $b_i \succ b_k$.

Besides the dominance relation between branches, it also exists among three or more branches. Generally, for branches $b_1, b_2, \cdots, b_{k-1}$, and b_k in program P', a set of branches, $\{b_1, b_2, \cdots, b_{k-1}\}$, is said to dominate b_k, when $\{b_1, b_2, \cdots, b_{k-1}\}$ is executed with any input of P', b_k must also be executed, denoted as $\{b_1, b_2, \cdots, b_{k-1}\} \succ b_k$. Although the dominance relation commonly exists among three or more branches, it is extremely time-consuming and difficult to analyze the dominance relation among them, not to mention listing all the dominance relations among them. This is because many additional combinatorial searches have to be conducted so as to list all the dominance relations among three or more mutant branches, and the complex combinatorial conditions further increase the difficulty of identifying such dominance relations. As a result, to reduce the number of mutants on the premise of complete dominance relations is not applicable. Instead it is well known that analyzing the dominance relation between mutant branches is relatively simple and time-effective. Although a very small portion of redundant mutants will survive after analyzing the dominance relations between mutant branches, they do not worsen the effectiveness of mutation testing at all. As a result, we consider the dominance relation only between mutant branches in this paper.

For two mutant branches b_i and b_j in P', if b_i does not dominate b_j, and b_j does not dominate b_i, it is said that there is no dominance relation between b_i and b_j, denoted as $b_i \| b_j$. Therefore, for two branches, b_i and b_j, in P', their dominance relation can be only one of the following three cases: $b_i \succ b_j$, $b_j \succ b_i$, and $b_i \| b_j$.

An interesting observation from the above example is that mutant branch 12 (b_{12}) dominates 18 (b_{18}), and mutant branch 18 (b_{18}) dominates 12 (b_{12}). This is, the above two mutant branches dominate each other. As a common rule, we select the mutant branch which will be executed earlier (e.g., $b_{12} \succ b_{18}$ in the above example), to potentially improve the computing efficiency. Similarly, the above rule will be applied to b_{13} and b_{14} ($b_{13} \succ b_{14}$ and $b_{14} \succ b_{13}$). Additionally, when two mutant branches dominate each other, it is possible that one branch is easier to be covered than the other for an automated testing tool, and remaining the mutant branch that is easier to be covered generally saves cost in generating a test datum.

Further, it is worth noting that the number of non-dominated mutant branches is the same, no matter which dominance relation is chosen. Besides, the chosen dominance relation does not affect the capability of a test suite in

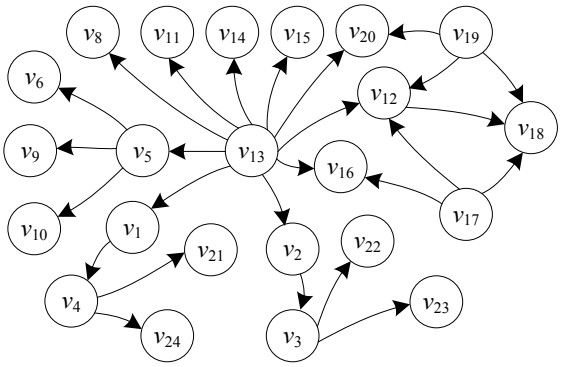

Figure 3: The dominance relation graph.

detecting faults. The reason is provided as follows. For two mutant branches that dominate each other, if we select a mutant branch as the non-dominated one, a test datum that covers the non-dominated mutant branch will also cover the dominated one. As covering a mutant branch equals to killing its corresponding mutant in weak mutation testing, the test datum can detect the faults corresponding to both mutant branches.

Based on Definition 2, all the dominance relations between each pair of mutant branches in Table 3 are provided as follows:

$b_1 \succ b_4$
$b_4 \succ b_{21}, b_{24}$
$b_2 \succ b_3$
$b_3 \succ b_{22}, b_{23}$
$b_5 \succ b_6, b_9, b_{10}$
$b_{12} \succ b_{18}$
$b_{13} \succ b_1, \cdots, b_6, b_8, \cdots, b_{12}, b_{14}, b_{15}, b_{16}, b_{20}$
$b_{17} \succ b_{12}, b_{16}, b_{18}$
$b_{19} \succ b_{12}, b_{18}, b_{20}$

3.3. Reducing Mutants

In view of the one-to-one mapping relation between a mutant and its mutant branch in P', this subsection only considers the set composed of all the mutant branches, denoted as B. The number of mutant branches is reduced after dominance analysis, and mutants corresponding to the non-dominated branches will remain.

Definition 3 Non-dominated branch

For an element b_i in B, it is called as a mutant branch. For $\forall b_j \in B$, if either $b_i \succ b_j$ or $b_i \| b_j$ is held, b_i is a non-dominated mutant branch. A set composed of all the non-dominated mutant branches is called the non-dominated branch set, denoted as B^{nd}.

Theorem 1: Mutants corresponding to B^{nd} are the ones after reduction.

Theorem 1 tells that seeking B^{nd} is of considerable importance to reduce mutants. In order to show the dominance relation more intuitively, we will introduce the definition of dominance relation graph as follows.

Definition 4 Dominance Relation Graph

The dominance relation graph of P' is a directional graph, denoted as $D(P') = \{V(B), E(B)\}$, where $V(B)$ is a vertex set, $E(B)$ is a directional edge set, and each mutant branch is represented as a vertex in $V(B)$. If the dominance relation exists between a pair of branches in B, e.g., $b_i \succ b_j$, there will be an edge from v_i to v_j, i.e., $< v_i, v_j > \in E(B)$.

Figure 3 is the dominance relation graph constructed from the dominance relations obtained in the last subsection. In the figure, there are 23 vertices (the other three correspond to infeasible mutant branches), $v_1, v_2, \cdots, v_6, v_8, v_9, \cdots, v_{24}$, and 26 edges with each representing the dominance relation in subsection 3.2. Figure 3 also reveals the transitivity of the dominance relation, e.g., $v_{13} \succ v_4$ is held since there exist edges $< v_{13}, v_1 >$ and $< v_1, v_4 >$.

In graph theory, in-degree and out-degree are two important indicators of the connectivity among vertices [49]. For v_i in $D(P')$, its in-degree indicates the number of branches in B that dominate b_i (the corresponding mutant branch

of v_i); on the contrary, its out-degree represents the number of branches in B that are dominated by b_i.

Theorem 2: Mutant branches corresponding to the vertices with zero in-degree form B^{nd}.

From Theorems 1 and 2, mutant branches corresponding to the vertices with zero in-degree in $D(P')$ are non-dominated ones, and mutants corresponding to those non-dominated mutant branches remain.

In Figure 3, vertices with zero in-degree are v_{13}, v_{17} and v_{19}. Therefore B^{nd} is composed of mutant branches corresponding to v_{13}, v_{17}, v_{19}, and the predicates of these non-dominated mutant branches are "$if((a == b)$!= $(\sim a == b))$", "$if((a == b)$!= $(a <= b))$", and "$if((a == b)$!= $(a >= b))$" (b_{13}, b_{17} and b_{19} in Table 3). Clearly, the original 23 (mutant branches 7, 25 and 26 in Table 3 are infeasible) mutants are reduced to 3.

As aforementioned in section 3.2, we choose $b_{12} \succ b_{18}$ and $b_{13} \succ b_{14}$ in the above example, although $b_{18} \succ b_{12}$ and $b_{14} \succ b_{13}$, where b_{12} is a dominated mutant branch ($b_{13} \succ b_{12}$, $b_{17} \succ b_{12}$ and $b_{19} \succ b_{12}$) and b_{13} is a non-dominated one. If $b_{18} \succ b_{12}$ and $b_{14} \succ b_{13}$ are chosen, we will get another graph by only swapping v_{12} with v_{18}, and v_{13} with v_{14} in Figure 3, respectively. In such case, b_{18} will be reduced, and b_{14} will be an element in B^{nd} instead of b_{13}. However for such dominance relation, although different choices might result in different mutant branches that remain, the number of non-dominated mutant branches is the same, i.e., the reduction rate remains unchanged.

The above illustrations and results suggest that our method can greatly reduce the number of mutant branches though its effectiveness needs further validation.

3.4. The Steps of Reducing Mutants

The steps of the proposed method for mutant reduction are as follows:

Step 1: Perform mutation operators on P to obtain the mutated statements.

Step 2: Construct mutant branches based on the statements before and after mutation, and then form the new program, P', according to subsection 3.1.

Step 3: Determine all the dominance relations between each pair of mutant branches.

Step 4: Construct the dominance relation graph $D(P')$ as described in subsection 3.3.

Step 5: Seek vertices with zero in-degree to form B^{nd}.

The time complexity of the above method depends on the step with the highest complexity in the above 5 steps. As defined in subsection 3.1, the number of mutants (mutant branches) is N. In step 1, to obtain N mutated statements, the mutation operation should be executed N times. To construct mutant branches in step 2, each mutated statement will be combined with its original one, i.e, the number of combinations is N. To identify all the dominance relations between each pair of mutant branches in Step 3, in the worst case, each of the N mutant branches will be compared with the other $(N-1)$ ones, so the biggest number of combinatorial search is $N \times (N-1)$. If $D(P')$ is stored in a matrix $Dom_{N \times N}$, the complexity of both Steps 4 and 5 are N^2. From the above analysis, we can obtain that the time complexity of our method is $O(N^2)$.

To identify dominance relations, we need to do the combinational search among mutant branches, which seems to have greatly increased the time complexity. However, as a dominated mutant branch will be deleted from B once it is found and therefore the number of branches in B becomes smaller, which in turn will reduce the time complexity in searching dominance relations and will obtain an acceptable time complexity of the combinational search process.

To obtain the non-dominated mutant branches, first, we choose a mutant branch b_i from B according to the time order in which a mutant branch enters B, and determine whether there is the dominance relation between b_i and any remaining mutant branch. If yes, delete all the dominated mutant branch(es) from B. Then we choose the next mutant branch from the set, B, and perform the above same process. By repeating the above process until no dominated mutant branches are left in the set, we will finally get the set B^{nd}.

4. Experiments

This section performs an experimental study to validate the effectiveness of the proposed method. First, the research questions are raised. Then, the experimental process is given. Finally, the experimental results on benchmark programs and open-source classes are analyzed.

4.1. Research Questions

Our purpose is to reduce mutants for the weak mutation testing. To this end, our experiment aims to answer the following questions:

RQ1: Can the proposed method effectively identify redundant mutants? This will be answered by the reduction rate, i.e., the ratio of the number of reduced mutant branches to the number of the total feasible mutant branches.

RQ2: Is the proposed method cost-effective to generate test data after reduction? This is reflected by the comparison on the time consumption between two programs before and after mutant reduction when using the same method to generate test data.

RQ3: Are test data yet effective after mutant reduction? This is illustrated by the mutation score achieved by the generated test data.

RQ4: How about the capability of test data generated after mutant reduction in detecting injected faults? The number of test data needed to achieve a desired mutation score is used to reflect the capability of test data in killing mutants.

4.2. The Experimental Process

Our experiments are done under the following environment: Intel(R) Core(TM)2 Duo CPU E7400 @ 2.80GHz, 2.79GHz, 2.0 GB ROM, Microsoft Windows XP SP3 operating system, and Eclipse 3.4 integrated development environment (IDE) with MuClipse 1.3 plug-in. MuClipse provides 15 method-level mutation operators and a bridge between the MuJava mutation engine and Eclipse IDE. By MuClipse, we generate mutants and execute mutation testing automatically, and compare the output of each mutant with that of the original one in Eclipse IDE. We extract the original and the mutated statements from MuClipse log to construct the mutant branches. The original program, P, is processed as follows: (1) the new program, P' (before reduction), is formed by fusing all the mutant branches into P manually, and (2) another new program, P'' (after reduction), is formed by fusing all the mutant branches after reduction into P manually.

Up to present, we have not found any existing tool that can generate mutant branches, form the transformed program, and especially, identify the dominance relations between each pair of mutant branches automatically. In our experimental study, mutant branches are first automatically formed based on the file, *mutation_log*, which is created by MuClipse when generating mutants. Then, these mutant branches are manually incorporated into the original program to construct the new program. Finally, manual analysis is performed to identify the dominance relation. Once all the dominance relations are identified, the dominance relation graph can be constructed automatically, and non-dominated branches will be obtained based on the vertices with zero in-degree.

Although it is feasible to automatically analyze and determine the dominance relation between mutant branches, the accuracy resulting from such an automatic way is not usually high. In fact, the dominance relation is semantic between mutant branches, which can be accurately obtained by manually semantic analysis. Unfortunately, it is difficult to form these semantic rules for further automatic analysis. Some would argue that the dominance relation can be detected by executing mutant branches with test data. However, the accuracy of this method highly depends on the number and the distribution of these test data, and to get enough test data with a good distribution is extremely difficult. As a result, we get the dominance relation between mutant branches manually instead of automatic analysis in this paper. Although our manual analysis is a bit time-consuming in these experiments, it is acceptable when considering the high accuracy of identified dominance.

Figure 4 shows the instrumented statements, "$if(a == b)\{trian = trian + 1;\}$", in Triangle, where (a) is P' obtained by fusing 23 mutant branches into P at appropriate positions, and (b) is P'' obtained by only fusing the non-dominated mutant branches. Array A in the figure indicates whether a mutant branch is covered or not, and each array index of array A is associated with a number in Table 3. All the instrumented branches in P' and P'' should be covered in our experiments.

We generate test data by the random approach. The random generation approach used in our experiments is composed of the following steps. First, the input domain of the program under test is determined. For example, the input of Triangle is (a, b, c), where a, b, c∈[-10, 9] which indicates that the value of each element can be sampled from [-10, 9]. Second, test data are generated by randomly sampling the input domain to cover the mutant branches before (after) reduction in P' (P''). The above steps are implemented in Java shown in Figure 4 (c).

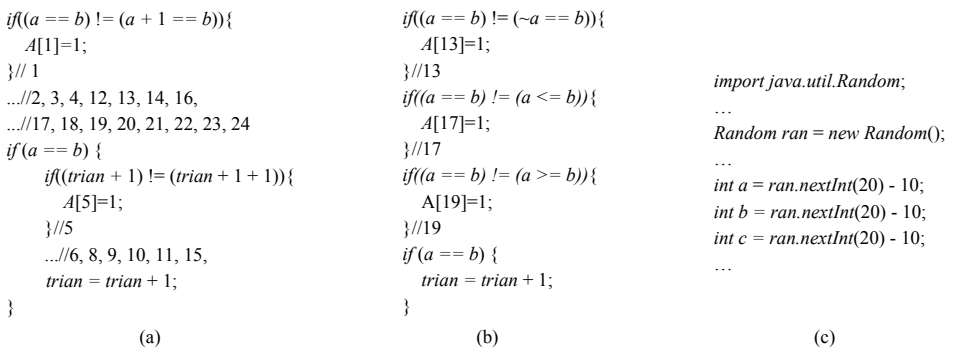

Figure 4: Instrumentation and test data generation, where (a) and (b) are transformed programs after inserting the mutant branches before and after reduction, respectively; and (c) is the test input sampling when generating test data.

Note that the main objective of this paper is to reduce mutants, and we aim to achieve that the mutants after reduction can be as effectiveness as the original ones, i.e. test data that cover all the mutant branches after reduction can also cover all the mutant branches before reduction. When comparing the effectiveness and the efficiency in generating test data, it is possible to select any reasonable method for test data generation. In this paper, the random sampling method is applied. Although there exist methods or tools such as Randoop for test data generation, test data generated by Randoop are not coverage-adequate, suggesting that additional test data must be added. Furthermore, the time spent in generating test data before and after mutant reduction should be calculated to evaluate the performance of the proposed method. Unfortunately, test data generated by Randoop is conducted in a given time. Although existing tools are not used when generating test data, the random method accords with the principle of random testing.

When generating test data to cover mutant branches in B (B^{nd}), first, we choose a mutant branch from B (B^{nd}), and then generate a test datum by randomly sampling the input domain until the test datum has covered the mutant branch. Taking mutant branch 5 (see $A[5]$ in Figure 4 (a)) as an example, to generate a test datum that covers it, we sample the input domain, $[-10, 9]^3$, for a number of iterations until a test input, $(2, 2, 3)$, has covered it. In this way, we generate the test datum that covers mutant branch 5. The above process is repeated until all the mutant branches in B (B^{nd}) have been covered.

It is clear that if the generated test data always first target the non-dominated branches, the results will be the same, no matter if it is before and after reduction. Nevertheless, the possibility of always targeting the non-dominated branches in B before reduction is very small, especially when there is a large number of mutant branches in B.

4.3. The Experimental Results and Analysis

To answer the questions raised in subsection 4.1, our method is applied to test ten benchmark programs and six classes from open-source projects.

4.3.1. Benchmark Programs

Ten programs in Table 4 are selected for the experiments. Among these programs, J1 is from [2], [4], [30] and [31]; J2, J3 and J4 are classical mutation testing programs appeared in [3]; J5 and J6 are example programs in [50], and J5, J6 and J7 are experimental programs in [17]; J8, J9 and J10 are from [51]. All these are programmed in Java.

(1) Regarding RQ1

Table 5 lists the reduction rates of the proposed method, where "Total" refers to the number of mutant branches in the new program, "Infeasible" represents the number of infeasible mutant branches, and "Remaining" is the number of feasible mutant branches after reduction by our approach.

From Table 5, we have two observations: (1) the ten programs achieve 80.59% reduction rate on average. For each program in Table 5, the dominance relation commonly exists not only between mutant branches constructed from the same statement, but also between those constructed from different statements. This explains the high reduction rates in Table 5; (2) the reduction rates vary from 73.27% to 92.22%, with the highest reduction rate from J5, and the lowest

Table 4: Information of benchmark programs.

ID	Name	LOCs	Methods	Function
J1	Triangle	35	1	Return the type of a triangle with three integer inputs.
J2	Euclid	12	1	Euclid's algorithm to find the greatest common divisor of two integers.
J3	Mid	26	1	Return the mid value of three integers.
J4	Bubble	16	1	Bubble sort algorithm.
J5	TrashAndTakeOut	26	2	Not reported.
J6	Cal	50	2	Calculate the number of days between two dates in the same year.
J7	Bank Account	34	3	Simulate bank account deposit and withdrawal services.
J8	Smoke Detector	40	1	Detect the current room smoke level by double inputs.
J9	Vending Machine	112	4	Vend some products through inserting coins, choosing items and getting changes.
J10	Sort Code	70	4	Validate the UK bank sortcode of the form XX-XX-XX, where X is a digit.

Table 5: Reduction of the proposed method.

ID	Mutant branches			Reduction rate (%)	Before reduction (P+M)		After reduction (GZY)		Speedup
	Total	Infeasible	Remaining		Test data	Time consumption (ms)	Test data	Time consumption (ms)	
J1	325	22	67	77.89	33.8	4.334284	23.6	0.650308	6.665
J2	57	11	10	78.26	5.7	0.439317	3.8	0.220242	1.995
J3	115	14	27	73.27	16.5	3.840068	9.8	0.699357	5.491
J4	92	5	16	81.61	5.8	10.002405	4.2	3.656434	2.736
J5	112	22	7	92.22	6.8	1.394492	4.7	0.297708	4.684
J6	316	43	54	80.22	25.7	169.806308	17.5	5.194644	32.689
J7	97	9	14	84.09	13.4	1.010467	9.5	0.329932	3.063
J8	175	19	33	78.85	11.8	2.028524	9.8	0.661253	3.068
J9	492	35	95	79.21	44.5	4698.295507	28.4	189.996382	24.728
J10	209	21	37	80.30	16.9	6.657047	12.5	1.686543	3.947
	Avg.			80.59					8.907

Table 6: The effectiveness of the generated test data.

ID	Mutants	Equivalent mutants	Before reduction (P+M)		After reduction (GZY)	
			Killed	Mutation score (%)	Killed	Mutation score (%)
J1	325	40	282	98.95	280	98.25
J2	57	12	45	100.00	45	100.00
J3	115	18	97	100.00	97	100.00
J4	92	7	85	100.00	85	100.00
J5	112	29	82	98.80	82	98.80
J6	316	43	273	100.00	266	97.44
J7	97	13	83	98.81	83	98.81
J8	175	20	155	100.00	155	100.00
J9	492	36	449	98.46	446	97.81
J10	209	40	168	99.41	168	99.41
Sum	1990	258	1719	Avg. = 99.44	1707	Avg. = 99.05

from J3. The reasons why the reduction rates vary among programs are given as follows. Different mutant branches are constructed from different programs. These mutant branches have different semantic relations for different programs, resulting in different dominance relations. A high reduction rate of program J5 is due to more dominance relations between each pair of its mutant branches. In contrast, the reason why program J3 has a low reduction rate is that there are relatively fewer dominance relations between each pair of its mutant branches.

(2) Regarding RQ2

In Table 5, time is employed to reflect the improvement of efficiency in generating test data after mutant reduction, which is more intuitive than other metrics such as the number of test data to demonstrate the efficiency. For example, a test data generation with less time is naturally to be regarded as more efficient. As a result, we compare the time consumption in generating test data after mutant reduction (GZY, Gong, Zhang, and Yao) to that before reduction (P+M, Papadakis and Malevris).

Note that the time consumption in generating test data is only a part of the cost in our method, which includes the time consumption spent in identifying dominance relations, determining non-dominated mutant branches, and generating test data. Furthermore, as we identify the dominance relation and reduce mutants manually, it is very difficult to give the time consumption in performing the manual tasks for each program. In fact, the time consumption in performing the manual tasks is closely related to the familiarity and skill of a tester, and a skilled tester who is familiar with the program often spends little time in performing the manual tasks. The complexity of the program and the constructed mutant branches is another factor that impacts the time consumption. To be specific, several seconds are required to identify a dominance relation between mutant branches constructed from the same original statement, such as $b_{19} \succ b_{20}$ in Table 3. Besides, tens of seconds are required to identify a dominance relation between mutant branches constructed from different original statements. Furthermore, if there is more complex code between mutant branches in the formed program, to identify the dominance relation will be very time-consuming, and the corresponding time consumption will be in the magnitude of minute. For a program under test, several hours or more time will be taken to identify all the dominance relations, e.g., the time consumption of J1 in Table 4 is over six hours. In practical testing, the manual process of a program is generally conducted by more than one tester. Therefore there is a difficulty in presenting the unified time consumption. Based on the above reasons, the time consumption of manual tasks is not provided any more in Table 5. "Speedup" which represents the ratio of the time consumption in generating test data before (P+M) to after reduction (GZY) is given in Table 5, without accounting for the cost spent on the manual tasks.

From Table 5, we have the following observations: (1) the ten programs obtain a speedup as high as 8.907 on average, due to the fact that over 80% mutant branches are reduced, and the newly formed programs are greatly simplified after reduction; (2) however, the speedups are different for different programs. For example, J6 and J9 get high speedups of 32.689 and 24.728, respectively. In addition to the high reduction rates of the two programs, reduced

Table 7: The number of test data needed for different mutation scores.

ID	Before reduction (P+M)				After reduction (GZY)			
	30%	60%	90%	Top	30%	60%	90%	Top
J1	6	8	24	34	2	4	12	24
J2	1	2	5	6	1	2	3	4
J3	2	7	14	16	2	4	7	10
J4	1	3	5	6	1	1	3	4
J5	1	4	6	7	1	2	3	5
J6	1	11	22	26	1	5	13	17
J7	4	5	10	13	1	3	6	9
J8	1	6	10	12	1	3	5	10
J9	4	12	28	45	3	10	16	28
J10	4	8	14	17	1	3	6	12

complexity of these programs is another reason for their high speedups.

(3) Regarding RQ3

To validate the effectiveness of test data generated after reduction, in strong mutation testing, we run all the mutants (without reduction) against test data generated before and after reduction. Table 6 lists the experimental results, where "killed" represents the number of killed mutants after (before) mutant reduction and "Mutation score (%)" represents the mutation score by the test data generated after (before) mutant reduction. In strong mutation testing, killing a mutant is more difficult than in weak mutation testing. Therefore, the number of equivalent mutants in Table 6 are more than the number of infeasible mutant branches in Table 5.

Although the number of equivalent mutants of programs J1-J7 in Table 4 can be obtained from the previous work of other researchers, they are not applicable in our approach. This is because we either employ a different programming language or use a different mutation tool from those used in the previous work, which results in a different numbers of equivalent mutants for the same program. On this circumstance, we identify all the equivalent mutants of each program manually, where the identification process is as follows. After running a series of test data on all the mutants of a program, survived mutants are semantically analyzed manually one by one in the "Compare Mutants" panel of MuClipse, and mutants that cannot be functionally differentiated from the original program are equivalent mutants. Mutants not killed by random chance have generally a high probability, and can be removed, which is helpful to improve the accuracy of manual analysis. As a result, our process of identifying the equivalent mutants has a high confidence.

As shown in Table 6, the mutation scores achieved by test data generated before and after reduction are 99.44% and 99.05%, respectively. The reason why a slight change occurs is as follows. Although our method can identify many dominated mutant branches. It also mistakenly takes seldom non-dominated mutant branches as dominated ones in the manual analysis process for dominance relation identification. Nevertheless, from Table 5, we can see that our method obtains a higher efficiency, where the average speedup is 8.907. From this point of view, it appears that there is an improvement on the speedup achieved by our method, however, one needs to take into account the additional time spent in producing the dominance relation set.

(4) Regarding RQ4

To illustrate the capability of test data generated after mutant reduction by our method in detecting the injected faults (mutants), for each program before and after reduction, the number of test data required to achieve the desired mutation scores are listed in Table 7. In the table, "Top" represents the number of test data required to reach the highest mutation score (which can be found from Table 6), and "30%" means the number of test data needed to reach the mutation score of 30%. The similar definition also applies to "60%" and "90%".

We have two observations from Table 7: (1) for each program, no matter whether its mutants are reduced or not, the number of test data needed increases with the desired mutation scores (30%, 60%, 90% and Top). This is because more test data are needed to kill increasing mutants, so as to achieve higher mutation scores; (2) for the same program, the number of test data needed to reach the same mutation score after reduction is less than that before reduction, which is due to the fact that the number of mutants needed to kill is less after reduction.

Table 8: Description of classes from open-source projects.

ID	Class	LOCs	Total methods	Tested methods	Mutants	Source
I1	WordUtils	173	12	2	243	org.apache.commons.lang3.text
I2	DurationFormatUtils	365	9	3	576	org.apache.commons.lang3.time
I3	HelpFormatter	416	39	4	477	org.apache.commons.cli
I4	NumberUtils	636	47	33	1406	org.apache.commons.lang3.math
I5	UnixCrypt	311	12	9	1097	org.apache.commons.codec.digest
I6	Md5Crypt	107	7	2	158	org.apache.commons.codec.digest
	Sum	2008	126	53	3957	

Five programs, J1, J3, J6, J9 and J10, are selected to show the change of the mutation scores along with the number of test data, which can be found in Figure 5. Since these programs generate many test data before and after reduction, the change cannot be intuitively reflected in Table 7. For the other five programs, J2, J4, J5, J7 and J8, as they generate a relatively smaller number of test data, so the change can be directly reflected in Table 7. Based on the above reason, we only graphically display the change of the mutation scores along with the number of test data for programs J1, J3, J6, J9 and J10.

For all the programs shown in Figure 5, the mutation scores increase rapidly before reaching 80% (or even 90% in some cases), and then increase slowly from 80% to the Top. For J1, before mutant reduction, only 12 test data are needed to achieve a mutation score of 80%. However, from 80% to Top (98.95%), 22 test data are needed; after mutant reduction, to achieve a mutation score of 80%, only 9 test data are needed, while from 80% to Top (98.25%), 15 test data are needed. This suggests that it is difficult to boost the mutation score greatly when it has reached a specific value (such as 80% or 90%), and there exist a few mutants that are difficult to kill.

Figure 5 also illustrates that, for each program, the same number of test data generated after mutant reduction usually achieves a higher mutation score. For example, five and twelve test data of program J10 get 50% and 70% mutation scores, respectively, before reduction. However, the same number of test data generated after mutant reduction obtains over 90% and Top (99.41%) mutation scores, respectively. The reason is that the number of test data needed to generate after reduction is reduced, and the capability of the generated test data in killing mutants is improved in the meanwhile.

The following conclusions can be drawn from the above experiments: (1) the proposed method can significantly reduce the number of mutants; (2) the time consumption for generating test data is shortened; (3) test data are effective for both weak and strong mutation testing; and (4) our approach generates a less number of test data. Therefore, the proposed method achieves an improvement for weak mutation testing.

In the experiments, we find that if a non-dominated mutant branch is covered by a test datum, its dominated ones are also covered. However, if a non-dominated mutant branch is not covered, it does not necessarily mean that its dominated ones cannot be covered. In fact, some or even all the dominated ones are covered, although the non-dominated one is not covered. That is to say, a non-dominated mutant is more difficult to kill than its dominated ones.

4.3.2. Classes from Open-source Projects

Six classes in Table 8 are selected from five open-source projects (`http://commons.apache.org`) to further validate our approach. The total LOCs of these classes is 2008. For 53 out of 126 methods, mutation testing is performed, and 3957 mutants are generated. Test data are generated to cover all the mutant branches in P' (before reduction) and P'' (after reduction), respectively, and all test data are output in the form of JUnit assertions. For example, in "assertEquals(1, Triangle.getTri(3, 4, 5));", the first parameter, 1, represents the expected output (a triangle type) given input "(3, 4, 5)", and the second parameter, Triangle.getTri(3, 4, 5), is the actual output of the program under test.

The following observations are drawn from Table 9: (1) the six classes generate 3957 mutant branches, among which 476 are infeasible. After mutant reduction by our method, the number of non-dominated mutant branches is

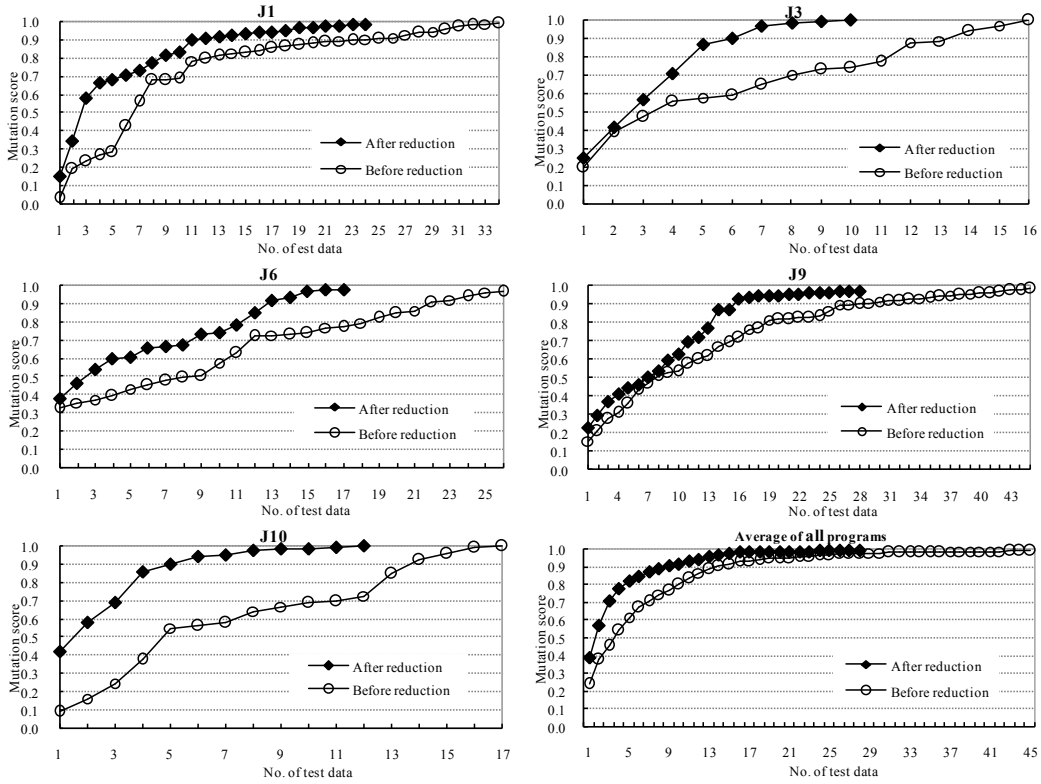

Figure 5: The change of the mutation scores along with the number of test data.

Table 9: Reduction of the proposed method.

ID	Mutant branches	Infeasible	Equivalent mutants	Remaining	Reduction rate (%)
I1	243	26	34	36	83.41
I2	576	52	65	83	84.16
I3	477	35	44	91	79.41
I4	1406	184	212	293	76.02
I5	1097	167	209	131	85.91
I6	158	12	12	30	79.45
Sum	3957	476	576	664	Avg. = 81.39

Table 10: The mutant branch coverage rates and the mutation scores achieved by test data generated before mutant reduction (P+M).

ID	Test data	Covered	Coverage rate (%)	Killed	Mutation score (%)
I1	9.7	213	98.16	203	97.13
I2	24.7	523	99.81	501	98.04
I3	23.9	434	98.19	426	98.38
I4	102.5	1208	98.85	1163	97.40
I5	9.4	903	97.10	859	96.73
I6	6.8	146	100	146	100
Sum	177.0	3427	Avg. = 98.69	3298	Avg. = 97.95

Table 11: The mutant branch coverage rates and the mutation scores achieved by test data generated after mutant reduction (GZY).

ID	Test data	Covered	Coverage rate (%)	Killed	Mutation score (%)
I1	4.8	211	97.24	199	95.22
I2	19.3	518	98.85	498	97.46
I3	18.9	429	97.06	420	97.00
I4	79.8	1178	96.40	1146	95.98
I5	5.7	901	96.88	842	94.82
I6	4.2	146	100	146	100
Sum	132.7	3383	Avg. = 97.74	3251	Avg. = 96.75

664 (shown as "Remaining" in Table 9), and the average reduction rate, which is as high as that of the benchmark programs shown in subsection 4.3.1, is up to 81.39%, where 2817 (calculated as $3957 - 476 - 664$) mutant branches are reduced; (2) among the six classes, the highest reduction rate is 85.91% from class I5, and the lowest is 76.02% from I4. The reason for the above observations is twofold: On one hand, each program achieves a relatively high reduction rate, and on the other hand, different programs would achieve different reduction rates.

Table 10 reports the mutant branch coverage rates and the mutation scores achieved by test data generated before mutant reduction (P+M). "Test data" in Table 10 represents the number of generated test data before reduction, while "Covered" ("Killed") means the number of covered (killed) mutant branches (mutants). Before reduction, 177.0 test data are generated, and 98.69% mutant branch coverage rate and 97.95% mutation score are achieved. It is also observed that, the lowest mutation score is 96.73% from class I5, and the highest is 100% from class I6.

In Table 11, after mutant reduction (GZY), the total number of generated test data of the six classes is 132.7. These test data cover 3383 feasible mutant branches (in Table 9, the total is $3957-476 = 3481$) with an average coverage rate of 97.74%, and kill 3251 mutants (from Tables 8 and 9, the number of non-equivalent mutants is $3957 - 576 = 3381$) with an average mutation score of 96.75%. Among them, the lowest mutation score is 94.82% from class I5, and the highest is 100% from class I6.

According to definition 3, if all the non-dominated mutant branches are correctly identified and covered by the generated test data, all the dominated mutant branches will be also covered in theory. However, identifying all the non-dominated mutant branches is very difficult, and closely related to a tester's familiarity with the program. Therefore, if a tester is unfamiliar with the program, one or more non-dominated mutant branches will be mistakenly regarded as the dominated ones. On this circumstance, the number of branches covered will decrease which is reflected by results in Tables 10 and 11.

Tables 10 and 11 suggest that: (1) the coverage rate and the mutation score after reduction drop by 0.95% and 1.20%, respectively, which are subtle compared with the reduction rate of 81.39% (in Table 9). The slight drop in coverage rate and mutation score is due to the manual analysis mistake made in identifying the dominance relation between mutant branches. The influences of the mistake are twofold: first, the dominated mutant branches being

Table 12: Comparison with mutant sampling

Sampling rate	ID	Remaining	Covered	Coverage rate (%)	Killed	Mutation score (%)
About 20%	I1	36	196	90.32	181	86.60
	I2	83	481	91.79	418	81.80
	I3	91	388	87.78	367	84.76
	I4	293	1104	90.34	1039	87.02
	I5	131	852	91.61	784	88.29
	I6	30	137	93.84	131	89.73
	Sum.	664	3158	Avg. = 90.95	2920	Avg. = 86.37
30%	I1	65	207	95.39	194	92.82
	I2	157	501	95.61	454	88.85
	I3	133	410	92.76	389	89.84
	I4	367	1126	92.14	1060	88.78
	I5	279	867	93.23	802	90.32
	I6	44	139	95.21	135	92.47
	Sum.	1045	3250	Avg. = 94.06	3034	Avg. = 90.51
40%	I1	87	210	96.77	195	93.30
	I2	210	510	97.33	471	92.17
	I3	177	416	94.12	398	91.92
	I4	489	1136	92.96	1084	90.79
	I5	372	894	96.13	820	92.34
	I6	59	141	96.58	139	95.21
	Sum.	1394	3307	Avg. = 95.65	3107	Avg. = 92.62

mistakenly regarded as non-dominated ones will result in a low reduction rate without changing the coverage rate; second, the non-dominated mutant branches being mistakenly regarded as dominated ones will cause a high reduction rate but a low coverage rate; (2) among the six classes, the maximal drops of the coverage rate and the mutation score are only 2.45% from I4 and 1.91% from I1 and I5, respectively, given that seldom mistakes are made in processing these two programs; (3) in addition, the average reduction of the number of test data is 44.3 dropping from 177.0 to 132.7, which is due to the fact that a great number of mutant branches are reduced.

Mutant sampling randomly selects a certain percentage of mutants to test, and the sampled mutants are those after reduction. There are some similar features between mutant sampling and our method, which are given as follows: (1) selecting (identifying) a small set of mutants to execute the mutation testing; (2) not for a small set of mutation operators. In fact, the mutants to be reduced by our method are from all the traditional operators of MuClipse. Furthermore, the distinct characteristics of our method are given as follows: (1) we transform mutants into mutant branches based on weak mutation testing; (2) instead of randomly sampling, we select mutants by dominance analysis.

To compare our method with mutant sampling, we first sample a certain number of feasible mutant branches from each class according to column "Remaining" in Table 9, and analyze the effectiveness of the sampled mutant branches, i.e. generating test data to cover the true parts of the sampled mutant branches and investigating how many mutant branches (mutants) are covered (killed) by the generated test data. Then, given the fact that the remaining mutant branches with our method is about 20% (the average reduction rate in Table 9 is 81.39%), we randomly sample 30% and 40% mutant branches to generate test data to cover (kill) all mutant branches (mutants).

In Table 12, "About 20%" means sampling the same number of mutant branches according to Table 9 (see "Remaining"). Table 12 reports that (1) when sampling "20%" mutant branches, the six classes get the same reduction rate as our method, whereas they achieve only 90.95% coverage rate and 86.37% mutation score, respectively, which are 6.79% and 10.38% lower than ours. The reason for low performances of mutant sampling lies in its inherent randomness, which will possibly cause a part of non-dominated mutant branches being reduced. (2) With the sam-

pling rates of 30% and 40%, random sampling has good performances in terms of coverage rate and the mutation score, but a low reduction rate, for which the reason is that the number of reduced non-dominated mutant branches decreases as the sampling rate increases. It is clear that a higher (more than 40%) sampling rate will achieve much better effectiveness, but much lower reduction rate (less than $1 - 40\%$). Compared with mutant sampling, our method can achieve a high reduction rate without much loss of the mutation score.

The experimental results from the six classes consist with those from the ten benchmark programs. By the proposed method, a great number of mutants are reduced; meanwhile, the mutation score remains high and each generated test datum can get an enhanced capability in detecting injected faults.

Although determining the dominance relation between mutant branches increases the complexities, our approach is suitable for programs with different scalabilities. To demonstrate this, we have done various experiments by applying our approach to programs with different sizes. As listed in Tables 4 and 8, our experimental subjects are from benchmark programs to classes from open-source projects, from programs with only one method to those with more than 30 methods, and from programs with ten LOCs to those with 636 LOCs. The experimental results show that our approach obtains a high reduction rate (see Tables 5 and 9), a great efficiency in generating test data after reduction (see Table 5), and a well-maintained effectiveness (see Tables 6, 7, 10, 11, and 12 as well as Figure 5).

5. Threats to the Validity

This section presents several threats to the validity of our experiments and the methods of addressing them.

Construct validity: Determining the dominance relation between mutant branches is of considerable importance to mutant reduction. In the experiments, we determine the dominance relation by manual analysis. It is clear that testers with different skills and familiarities with a program will give different analysis results for the same pair of branches. To reduce this kind of threats, we select skilled testers who are as familiar with the program as possible, and make sure they follow all the steps of the proposed method strictly in the experiments. Nevertheless, the above threats cannot be completely eliminated. To overcome this, we will further study how to automatically identify the dominance relation between mutant branches.

The time consumption is an important indicator to reflect the efficiency in generating test data. In the experiments, we describe the time consumption in the form of millisecond which is closely related to the configuration of our implementation environment such as the number and the type of processors, the operating system, etc. It is clear that environments with different configurations will have different time consumptions when fulfilling the same task. To alleviate this kind of threats, for each program before and after mutant reduction, we run the same task of generating test data for ten times in the same environment, then record all these time consumptions, and finally calculate the average time consumption of them.

Internal validity: The mutant reduction rate is the unique indicator to reflect the capability of our approach in reducing mutants. In order to calculate the mutant reduction rate, we need to know the number of feasible mutant branches and the remaining mutant branches after reduction. We can easily obtain the number of mutant branches after reduction. However, to determine the feasibility of a mutant branch, we need to employ appropriate approaches. In this paper, we utilize the method proposed in [42]. However, this method cannot guarantee that all the feasible mutant branches of a program are detected. To overcome this, we will investigate other more effective methods and this is left for our further work.

External validity: In the experiments, we utilize the random method to generate test data to cover mutant branches before and after reduction. Although the random method is highly intuitive and easy to implement, it is not an approach with the highest efficiency for generating test data. However, what we highlight in this paper is to propose an efficient approach to reduce mutants other than generate test data. As a result, we will not put much efforts at this stage to select the best method of generating test data. Nevertheless, we will explore other more effective methods to fulfil this task in our future work.

Additionally, although the preliminary experimental studies suggest that our approach is effective for a set of small programs, the above conclusion cannot be directly generalized to more complicated programs. To alleviate this kind of threats as much as possible, we select programs with different sizes and features when conducting the experiments. Further experiments on more complicated programs will be conducted in the future to validate the scalability of the proposed approach.

6. Conclusions

In mutation testing for complex software, there will be a large number of mutants being generated, which leads to a too high computational cost to be practically used. Reducing mutants is proved to be an effective way to improve the efficiency of mutation testing. However, there have been not yet efficient approaches available, which are able to reduce a large number of mutants and maintain a high mutation score at the same time.

We focus on mutant reduction in weak mutation testing, and propose a novel method of reducing mutants based on dominance relation. In our method, a mutant branch is first constructed by combining the statements before and after mutation, and a new program is formed by integrating all the mutant branches into the original program. Then, dominance analysis is conducted in the new program. Finally, a dominance relation graph is built, and the non-dominated mutants are identified from vertices with zero in-degree. The experimental results show that the proposed method is competent in reducing mutants. After mutant reduction, time spent in generating test data is shortened, while the generated test data are effective in killing all the mutants. The capability of each test datum in detecting the injected faults is also proved to be strong. As a result, we conclude that the efficiency of mutation testing is greatly improved by our method.

It should be pointed out that the proposed method is only suitable for reducing mutants generated by traditional (method-level) mutation operators and not suitable for class-level operators. Therefore, it is of necessity to study effective methods to reduce mutants generated by the class-level mutation operators. Additionally, in the proposed method, there is a necessity to manually analyze the source code, which will cause scalability issues and therefore restrict the practical applications of our method. Therefore, it is of considerable necessity to develop appropriate solutions to automate the proposed method, and this is part of our future work. To fulfill this task, we first mutate the statement to construct the mutant branches, and insert each mutant branch before its original statement. Then, we employ statistical methods to analyze the probability of covering mutant branch b_j when b_i is covered, $prob(b_j|b_i)$, and conclude that b_i dominates b_j when $prob(b_j|b_i) = 1$. Finally, we determine all the dominated mutant branches by the above statistical approach. The automated analysis of the dominance relation between mutant branches will be a significant research task in our near future work, and related work such as Kurtz et al. [46], Kintis and Malevris [52], Offutt and Pan [53], and Zhang et al. [54] will be of great help in achieving this goal. Furthermore, we find that different branches have different numbers of dominated branches. Based on this observation, if the priority of generating a test datum can be determined, the quality of test data will be further improved, which is one of our future research directions. Finally, the implementation of a prototype system based on the proposed method and the evaluation on larger software projects are also in the plan.

7. Acknowledgement

This work is jointly supported by National Natural Science Foundation of China (No. 61375067 and 61203304), Natural Science Foundation of Jiangsu Province (No. BK2012566). We would like to thank Dr. Edward C. Mignot (the formerly Professor of Shandong University) for polishing this paper.

References

[1] G. Fraser, A. Zeller. Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering, 2012, 38(2), 278-292.
[2] A. J. Offutt, J. Pan, K. Tewary, T. Zhang. An experimental evaluation of data flow and mutation testing. Software: Practice and Experience, 1996, 26(2), 165-176.
[3] Y. Jia, M. Harman. Analysis and survey of the development mutation testing. IEEE Transactions on Software and Engineering, 2011, 37(5), 649-678.
[4] Y. Jia, M. Harman. Higher order mutation testing. Information and Software Technology, 2009, 51(10), 1379-1393.
[5] W. E. Howden. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering, 1982, 8(4), 371-379.
[6] J. Chen, Y. Lu, X. Xie. Research on software fault injection testing. Chinese Journal of Software, 2009, 20(6), 1425-1443.
[7] J. H. Andrews, L. C. Briand, Y. Labiche. Is mutation an appropriate tool for testing experiments. Proceedings of International Conference on Software Engineering, 2005, 402-411.
[8] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering, 2006, 32(8), 608-624.
[9] F. Lammermann, A. Baresel, J. Wegener. Evaluating evolutionary testability for structure-oriented testing with software measurements. Applied Soft Computing, 2008, 2, 1018-1028.

[10] J. J. Dominguez-Jimenez, A. Estero-Botaro, A. Garcia-Dominguez, I. Medina-Bulo. Evolutionary mutation testing. Information and Software Technology, 2011, 53(10), 1108-1123.
[11] R. Just, M. D. Ernst, G. Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. Proceedings of Dagstuhl Seminar 13021: Symbolic Methods in Testing, abs/1303.2784, 2013.
[12] R. Just, G. M. Kapfhammer, F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. Proceedings of 23rd International Symposium on Software Reliability Engineering, 2012, 11-20.
[13] R. Just, G. M. Kapfhammer, F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis. Proceedings of IEEE 5th International Conference on Software Tesing, Verification and Validation, 2012, 720-725.
[14] G. Kaminski, P. Ammann, J. Offutt. Better predicate testing. Proceedings of 6th International Workshop on Automation of Software Test, 2011, 57-63.
[15] M. F. Lau, Y. T. Yu. An extended fault class hierarchy for specification-based testing. ACM Transactions on Software Engineering and Methodology, 2005, 14(3), 247-276.
[16] R. H. Untch, A. J. Offutt, M. J. Harrold. Mutation analysis using mutant schemata. Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, 1993, 139-148.
[17] M. Papadakis, N. Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal, 2011, 19(4), 691-723.
[18] M. Marre, A. Bertolino. Using spanning sets for coverage testing. IEEE Transactions on Software Engineering, 2003, 29(11), 974-984.
[19] A. S. Ghiduk, M. R. Girgis. Using genetic algorithms and dominance concepts for generating reduced test data. Informatica, 2010, 34(3), 377-385.
[20] D. Gong, G. Zhang, X. Yao. Mutant reduction based on the dominant relation, Abstract of 2nd Chinese Search Based Software Engineering (CSBSE'2013), 2013, 14-15.
[21] A. T. Acree. On Mutation. PhD Thesis, Georgia Institute of Technology, 1980.
[22] T. A. Budd. Mutation Analysis of Program Test Data. PhD Thesis, Yale University, 1980.
[23] A. P. Mathur, W. E. Wong. An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria. Technical Report, Purdue University, 1993.
[24] S. Hussain. Mutation Clustering. Master's Thesis, King's College London, 2008.
[25] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. Proceedings of Computer Software and Applications Conference, 1991, 604-605.
[26] A. J. Offutt, G. Rothermel, C. Zapf. An experimental evaluation of selective mutation. Proceedings of 15th International Conference on Software Engineering, 1993, 100-107.
[27] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engneering and Methodology, 1996, 5(2), 99-118.
[28] E. S. Mresa, L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability,1999, 9(4), 205- 232.
[29] X. Yao, M. Harman, Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. Proceedings of 36th International Conference on Software Engineering, 2014, 919-930.
[30] Y. Jia, M. Harman. Constructing subtle faults using higher order mutation testing. Proceedings of 8th IEEE International Working Conference on Source Code Analysis and Manipulation, 2008, 249-258.
[31] W. B. Langdon, M. Harman, Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. The Journal of Systems and Software, 2010, 83(12), 2416-2430.
[32] M. Polo, M. Piattini, I. Garcia-Rodriguez. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification and Reliability, 2008, 19(2), 111-131.
[33] M. Kintis, M. Papadakis, N. Malevris. Evaluating mutation testing alternatives: a collateral experiment. Proceedings of 17th Asia Pacific Software Engineering Conference, 2010, 300-309.
[34] M. Papadakis, N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. Proceedings of 3rd International Conference on Software Testing, Verification, and Validation Workshops, 2010, 90-99.
[35] R. A. DeMillo, A. J. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 1991, 17(9), 900-910.
[36] R. A. DeMillo, A. J. Offutt. Experimental results from an automatic test case generator. ACM Transactions on Software Engineering and Methodology, 1993, 2(2), 109-127.
[37] J. R. Horgan, A. P. Mathur. Weak Mutation is Probably Strong Mutation. Technical Report, Purdue University, 1993.
[38] A. J. Offutt, S. D. Lee. How strong is weak mutation. Proceedings of 4th Symposium on Software Testing, Analysis, and Verification, 1991, 200-213.
[39] A. J. Offutt, S. D. Lee. An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering, 1994, 20(5), 377-344.
[40] M. Papadakis, N. Malevris. Mutation based test case generation via a path selection strategy. Information and Software Technology, 2012, 54(9), 915 - 312.
[41] A. Bertolino, M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Transactions on Software Engineering, 1994, 20(12), 885-899.
[42] D. W. Gong, X. J. Yao. Automatic detection of infeasible paths in software testing. IET Software, 2010, 4(5), 361-370.
[43] J. H. Shan, Y. F. Gao, M. H. Liu, J. H. Liu, L. Zhang, J. Sun. A new approach to automated test data generation in mutation testing. Chinese Journal of Computers, 2008, 31(6), 1025-1034.
[44] P. Ammann, M. E. Delamaro, J. Offutt. Establishing theoretical minimal sets of mutants. Proceedings of 7th IEEE International Conference on Software Testing, Verification, and Validation. 2014, 21-30.
[45] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, L. Deng. Mutant subsumption graphs. Proceedings of 7th IEEE International Conference on Software Testing, Verification, and Validation. 2014, 176-185.

[46] B. Kurtz, P. Ammann, J. Offutt. Static analysis of mutant subsumption. Proceedings of 8th IEEE International Conference on Software Testing, Verification, and Validation. 2015, 1-10.
[47] B. H. Smith, L. Williams. An empirical evaluation of the MuJava mutation operators. Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques, 2007, 193-202.
[48] S. Segura, R. M. Hierons, D. Benavides, A. Ruiz-Cortes. Automated metamorphic testing on the analyses of feature models. Information and Software Technology, 2007, 53(3), 245-258.
[49] J. A. Bondy, U. S. R. Murty. Graph Theory with Application. American Elsevier, New York, 1976.
[50] P. Ammann, J. Offutt. Introduction to Software Testing. Cambridge: Cambridge University Press, 2008.
[51] P. McMinn. Evolutionary Search for Test Data in the Presence of State Behaviour. PhD thesis, The University of Sheffield, 2005.
[52] M. Kintis, N. Malevris. Using data flow patterns for equivalent mutant detection. Proceedings of 7th IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2014, 196-205.
[53] A. J. Offutt, J. Pan. Automatically detecting equivalent mutants and infeasible paths. Software Testing Verification & Reliability , 1997, 7(3), 165-192.
[54] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. Halleux, H. Mei. Test generation via dynamic symbolic execution for mutation testing. Proceedings of IEEE International Conference on Software Maintenance. 2010, 12-18.