Hindawi
Scientific Programming
Volume 2019, Article ID 4702345, 11 pages
https://doi.org/10.1155/2019/4702345

Research Article

Mutant Selecting According to the Nondominated Original Statements

Gongjie Zhang,[1] **Chunli Xie**,[1] **Yongquan Dong**,[1] **and Qiao Yu**[1,2]

[1]*School of Computer Science and Technology, Jiangsu Normal University, 221116 Xuzhou, China*
[2]*Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, 541004 Guilin, China*

Correspondence should be addressed to Gongjie Zhang; zhanggongjie@126.com

Received 16 March 2019; Revised 12 August 2019; Accepted 5 September 2019; Published 3 November 2019

Academic Editor: Giuseppe Scanniello

Copyright © 2019 Gongjie Zhang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Mutation testing is a technique for evaluating the quality of a test suite. However, the costly computation from a large number of mutants affects the practical application of mutation testing, and reducing the number of mutants is reasonably an efficient way for mutation testing. We propose a new method for reducing mutants by analyzing dominance between statements in the program under test. The proposed method only selects the mutants generated from the nondominated statements, and the mutants generated from the dominated statements are reduced. After applying the proposed method to nine programs, the experimental results show that our method reduces over 75% mutants and well maintains the mutation adequacy.

1. Introduction

Mutation testing is a fault-oriented testing technique first proposed by Hamlet [1] and DeMillo et al. [2]. In mutation testing, the program under test is mutated according to the given syntax rules (mutation operators), and each mutated version is a mutant. When a mutant differs from its original program in their outputs after executing against the same test case, the mutant is said killed by the test case. Specially, the mutants that function equal to its original program and cannot be killed by any test case are equivalent. Mutation score which evaluates the adequacy of a given test suite is calculated by the ratio of the number of killed mutants to the number of nonequivalent ones.

It has been demonstrated empirically that mutants can reflect real faults in the program [3–5]. Generally, mutation testing is used to evaluate the quality of a test suite [6] or aided to generate test suites with high ability of fault detection [7]. However, a large number of mutants are generated from the program under test result in a high cost, which hinders the mutation testing from wide application. And according to a recent survey [8], a large number of mutants, still today, are considered as one of the most important problems of mutation testing.

Practically, most faults in programs tend to locate in a few statements instead of all the statements. It seems that performing mutation operators on all the statements is unnecessary. For a given test suite, studies have shown that the more times the mutated statement is executed, the higher likelihood of killing the corresponding mutants [9, 10]. Inspired by this, we only focus on some statements (instead of all the statements) in the program and only select the mutants generated from these statements, thus reducing the cost of mutation testing.

In view of this, we hope to identify a subset of statements that the test suite that covers the ones will cover all the statements in the program. Consequently, we propose a new method for identifying the nondominated statements in the program. The proposed method only selects the mutants generated from the nondominated statements, and the other mutants will be reduced. The experimental results from nine programs suggest that our method can reduce a large number of mutants without significant loss in mutation adequacy.

The rest of this article is organized as follows: Section 2 summarizes the related work; the proposed method will be detailedly described in Section 3, including the definitions for identifying the nondominated statements and an

example for preliminary illustration; Section 4 evaluates our method through a series of experimental studies; and finally, Section 5 concludes our contributions.

2. Related Work

Focusing on saving the cost in mutation testing, in this section, we first state the existing methods for mutant reduction, and then the methods for optimizing mutation testing are discussed; since our method is based on correlation analysis, we finally summarize correlation analysis, especially in mutation testing.

2.1. Mutant Reduction. Reducing the number of mutants can save the cost in mutation testing. As summarized in the survey of Jia and Harman [11], the classical methods of mutant reduction including mutant sampling and selective mutation testing are very simple and practical, but they cannot achieve a high reduction rate while maintaining a high mutation adequacy; although high-order mutants (HOMs) can both reduce the number of mutants and represent complex real faults, the cost for generating HOMs is very expensive. Second-order mutants (SOMs), which reduce about half number of mutants, can be a compromise substitution for HOMs. Kintis et al. proposed strategies for combining SOMs based on the control relations among nodes in the control flow graph (CFG) of a program [12]. It is noticeable that many SOMs are not necessary to be generated since they are easy to kill.

It is possible to save some cost by eliminating the equivalent mutants. Yao et al. revealed the distribution features of equivalent mutants and stubborn mutants by manual analysis [13], and their work is beneficial for designing mutation operators to avoid generating the equivalent mutants. To tackle the problem of detecting equivalent mutants and generating test data to kill the nonequivalent ones in narrow spaces, Harman et al. analyzed the dependence relations between nodes and variables [14]. Hierons et al. applied program slicing to simplify the programs, thus reducing the effort in detecting equivalent mutants [15]. Compared with their methods, we only consider the nondominated statements (nodes), and it seems that the grain of our method is coarser than that of [14] but finer than that of [15]. However, it is reported that the percentage of equivalent mutants is only 10%–40% [16, 17].

We explore new method for reducing mutants, and our method only selects a small proportion of mutants generated from a subset of statements in the program.

2.2. Optimization of Mutation Testing. A mutant is weakly killed if its state immediately differs from the original program after executing the mutant. Although weak mutation testing seems less effective than strong mutation testing, according to the survey [8], weak mutation can save much execution time by not executing the code after the mutated statements.

To speed up weak mutation testing, Durelli et al. constructed virtual machine-integrated environment for mutation testing [18]. By using the high speed of noninterpretive system, Kim et al. developed a Java mutation system to save the cost in weak mutation testing [19]. Papadakis and Malevris proposed a transformation method for further improving weak mutation testing [20]. In their method, they transformed mutants into mutant branches and integrated all the mutant branches into the original program to form a new program, and covering the mutant branches indicates killing the corresponding mutants in weak mutation testing. However, a lot of mutant branches in the new program add its complexity. Moreover, they generate test cases to kill mutants by selecting paths to cover [7]. And many mutants cause a very large search space, thus increasing the cost in generating test cases.

For given test cases, Zhang et al. found that the more times a mutated statement executes, the higher probability of the corresponding mutants being killed [9]. According the observation, Zhang et al. prioritized test cases so as to earlier execute the test cases that are more likely to kill mutants [10]. And their method saves about half cost of mutation testing. Inspired by the above observation, we select mutants according to the original statements, and we expect that the selected ones can well maintain the mutation adequacy.

2.3. Correlation Analysis. Correlation analysis is very helpful for mutation testing. Shan et al. combined compound mutants according to the correlations among mutated statements of the same statement and generated test cases to kill the compound mutants [21]. Their method reduces the number of mutants and generates less number of test cases. Xu analyzed correlations among the mutants generated from the conditional statements and reduced about 20% mutants based on the control and equivalent relations between these mutants [22].

Subsumption was found between nodes by Kintis et al., when analyzing correlations among nodes in the CFG [12]. Kaminski et al. demonstrated that three mutation rules of ROR mutation operators subsume other four mutation rules of ROR [23]. Ammann et al. identified the redundant mutants by analyzing dynamic subsumption between mutants after executing the mutants against a given test suite [24]. Kurtz et al. further defined the subsumption as true subsumption, dynamic subsumption, and static subsumption, and described subsumption between mutants with a mutant subsumption graph (MSG) [25]. In the MSG, the root nodes subsume other nodes; that is, the test cases that kill the mutants corresponding to the root nodes can kill the subsumed mutants. Kurtz et al. automatically constructed the static MSG and generated test cases to kill the mutants corresponding to the root nodes [26], so as to save the cost in generating test cases.

In our previous research [27], we transformed the mutants into the mutant branches to reflect whether the corresponding mutants are weakly killed or not, and formed a new program by inserting all the mutant branches into the original program; and the nondominated ones, corresponding to the mutants after reduction, were obtained after analyzing the dominance relations among the mutant branches. Our previous method can further improve the

weak mutation testing by only covering the nondominated mutant branches, and the generated test cases can cover all the mutant branches, that is, weakly kill all the mutants.

Inspired by above researches, we only select the mutants generated from the nondominated statements in the program. The proposed method and our previous research have some similar features as follows: (1) they are based on dominance analysis among statements (mutant branches), (2) the nondominated statements (mutant branches) correspond to the mutants after reduction, and (3) the dominance relation graph is used to describe the dominance relations.

Moreover, the proposed method has some specific features as follows: (1) It is not necessary to construct the mutant branches or form a new program. (2) The dominance relations are analyzed among the statements in the original program instead of among the mutant branches in the formed program. Since the number of the original statements is significantly less than the number of mutant branches, it will be more efficient to analyze dominance relations among the original statements. (3) There may exist dominance relations among the selected mutants in our method. In our previous research, there is no dominance relation among mutants after reduction. (4) A node in the dominance relation graph represents an original statement and an edge is a dominance relation between statements. Differently, in our previous research, a node in the dominance relation graph is a mutant branch and an edge indicates a dominance relation between a pair of mutant branches.

3. The Proposed Method

For the purpose of reducing mutants, we first analyze the dominance relation between statements in the program, and the nondominated statements are obtained; then, we select the mutants according to the nondominated statements; finally, an example is presented as a preliminary illustrating.

3.1. Dominance Relation. Suppose that the program under test is P, and the set of statements that can be mutated in P is S. Applying the mutation operators to $s_i \in S$, $i = 1, 2, \ldots, |S|$, the set of mutants is denoted as M_i. Since the mutants in M_i are generated from s_i, s_i and the mutated statements of M_i have the same reachability, that is, for any test case $t \in T$, if t execute s_i in P, then t must execute all the mutated statements in M_i.

As known, reachability is the first condition of killing a mutant, and the others are necessity and sufficiency. For $s_i, s_j \in S$ and $s_i \neq s_j$, the generated mutant sets are M_i and M_j respectively. If s_i and s_j have the same reachability in P, the reachability of killing the mutants in M_i and M_j are the same. Therefore, the reachability conditions can be determined by the corresponding original statements.

Emphatically, correlations exist among the reachability conditions of statements in the program. As shown in Figure 1, if statement "$c > b$" (line 4) is executed, then "$a > b$" (line 3) must be executed. Similarly, if "mid $= c$" (line 6) is executed, statements "$a > b$" (line 3), "$c > b$" (line 4), and "$c > a$" (line 5) must be executed. This relation is called dominance relation, and its definition is as follows.

Definition 1 (dominance relation). Suppose that $s_i, s_j \in S$ are statements in P. For any $t \in T$, if s_i is executed, then s_j must be executed, we say that s_i dominates s_j, denoted as $s_i \succ s_j$. In this case, s_i is the dominating statement and s_j is the dominated statement.

The set of all the dominance relations between statements in S is called the dominance relation set, denoted as Dom. It should be noted that the concept of dominance relation has been previously defined for reducing the number of targets needed to cover [28]. Here, we use dominance relations to assist in selecting mutants.

Accordingly, if $s_i \succ s_j$, the probability of executing s_j is not less than that of executing s_i. The reasons are as follows: the test cases that execute s_i must also execute s_j, and other tests that cannot execute s_i may also execute s_j. And in consequence, the probability of executing the mutated statements of mutants in M_j is not less than that in M_i. Although satisfying the reachability does not indicate killing a mutant, we have a basic intuition that the more times the mutated statement is executed, the higher probability of killing the corresponding mutant.

3.2. Nondominated Statement. The code in Figure 1 has 12 mutable statements, denoted as $S = \{s_1, s_2, \ldots, s_{12}\}$. And the dominance relations between these statements are obtained by manual analysis, as listed in Table 1.

Since the execution probability of a dominated statement is not less than its dominating statement, based on the intuition in the previous section, it is intuitive that the mutants generated from a dominated statement can more probably be killed than that from its dominating statement.

In addition, a statement can be a dominating or a dominated one. We aim to find the statements that are not dominated by any statements, and such a statement is nondominated. The definition of nondominated statement can be given as follows.

Definition 2 (nondominated statement). Suppose $s_i \in S$ is a statement in P. If $\neg\exists s_j \in S, s_j \neq s_i$, such that $s_j \succ s_i$, s_i is a nondominated statement.

The set of all the nondominated statement is denoted as S^{nd}. There is an interesting phenomenon: for $s_i, s_j \in S$ and $s_i \neq s_j$, $s_i \succ s_j$, and meanwhile $s_j \succ s_i$. In this circumstance, we only reserve one of them, and the other will be removed. And as a common rule, we reserve $s_i \succ s_j$ and remove $s_j \succ s_i$, if $i < j$. As shown in Table 1, $s_{12} \succ s_1$ will be removed since $s_1 \succ s_{12}$.

3.3. Determining Nondominated Statements. The dominance relations between statements can be described in the dominance relation graph, and its definition is as follows.

```
(1) public static int getMid (int a, int b, int c) {
(2)    int mid;
(3)    if (a > b) {
(4)       if (c > b) {
(5)          if (c > a) {
(6)             mid = c;
(7)          } else {
(8)             mid = a;
(9)          }
(10)      } else {
(11)         mid = b;
(12)      }
(13)   } else {
(14)      if (b > c) {
(15)         if (a > c){
(16)            mid = a;
(17)         } else {
(18)            mid = c;
(19)         }
(20)      } else {
(21)         mid = b;
(22)      }
(23)   }
(24)   return mid;
(25) }
```

Figure 1: The program code of Mid.

Table 1: The dominance relations between the statements.

s_i	Statement	Line	Dominated statement
s_1	$a > b$	3	s_{12}
s_2	$c > b$	4	s_1, s_{12}
s_3	$c > a$	5	s_1, s_2, s_{12}
s_4	mid = c	6	s_1, s_2, s_3, s_{12}
s_5	mid = a	8	s_1, s_2, s_3, s_{12}
s_6	mid = b	11	s_1, s_{12}
s_7	$b > c$	14	s_1, s_{12}
s_8	$a > c$	15	s_1, s_7, s_{12}
s_9	mid = a	16	s_1, s_7, s_8, s_{12}
s_{10}	mid = c	18	s_1, s_7, s_{12}
s_{11}	mid = b	21	s_1, s_7, s_{12}
s_{12}	mid	24	s_1

Definition 3 (dominance relation graph). Dominance relation graph is a directional graph, denoted as $DG(S) = (V(S), E(\text{Dom}))$, where $V(S)$ is the vertex set of the statements in P and $E(\text{Dom})$ is the set of directional edges. For $s_i, s_j \in V(S)$, there is an directional edge $< s_i, s_j > \in E(\text{Dom})$ if and only if $s_i \succ s_j$.

According to Definition 3, $s_i \in S$ if and only if $s_i \in V(S)$, and $s_i \succ s_j \in \text{Dom}$ if and only if $< s_i, s_j > \in E(\text{Dom})$. In $DG(S)$, the in-degree of vertex s_i indicates the number of statements that dominate s_i, and the out-degree of s_i reflects the number of statements that are dominated by s_i. A vertex s_i with zero in-degree means that there is no $s_j \in S$ to make $s_j \succ s_i$ hold. Therefore, the vertices with zero in-degree in the dominance relation graph correspond to the nondominated statements.

Figure 2 is the dominance relation graph of the program in Figure 1. From Definition 3 and Figure 2, the set of nondominated statement of the program in Figure 1 is $S^{nd} = \{s_4, s_5, s_6, s_9, s_{10}, s_{11}\}$. For a test set T, if T can execute all the statements in S^{nd}, T will execute all the statements in S. The corresponding mutant sets of $s_4, s_5, s_6, s_9, s_{10}$, and s_{11} are $M_4, M_5, M_6, M_9, M_{10}$, and M_{11}, respectively. For T, if $s_i \succ s_j$, the execution times of the mutated statements in M_i are not more than that of the mutated statements in M_j. Accordingly, we only select the mutants generated from the statements corresponding to the vertices with zero in-degree.

After performing all the traditional (Method-level) mutation operators of MuClipse on the program in Figure 1, 115 mutants are generated, as listed in Table 2, including eighteen equivalent ones (the italic values in Table 2). Among the mutants in Table 2, 36 mutants generated from the nondominated statements will be selected to perform mutation testing.

3.4. Illustrative Example. Whether the selected mutants can well maintain the mutation adequacy will be preliminarily illustrated by an example.

The given test set T including eleven test cases, as listed in Table 3, and each test case is programmed in the form of JUnit assertion, assert XXX (expected, real). The parameter expected of the assertion is the expected value by executing the program against a given input, and *real* is the real value of the program by executing the same input. If the real value equals to the expected value, the assertion returns true. For example, assert Equals (9, Mid.getMid (9, 9, 2)) (t_1 in Table 3), the expected value by input $(9, 9, 2)$ is 9, and the real value is Mid.getMid $(9, 9, 2)$.

Moreover, the killed mutants by eleven test cases are listed in Table 3. From Table 3, we have following observations. (1) 24 mutants are killed by the given test cases, and the twelve mutants alive are equivalent (listed in Table 3). (2) The effective test set for the selected mutants is $T' = \{t_1, t_2, t_3, t_4, t_5, t_{10}\}$. Consequently, the mutation score of T is $24/(36 - 12) * 100\% = 100\%$; that is, T kills all the nonequivalent ones of the selected mutants.

Importantly, what we more care about is whether the selected mutants can be as adequate as all the mutants in evaluating a given test set, that is, how many mutants will be killed by the effective test cases of the selected mutants. Furthermore, we execute 115 mutants against T', and the killed mutant are listed in Table 4. As shown in Table 4, T' kills 94 nonequivalent mutants, and three nonequivalent ones alive are AOIS_14, LOI_6 and ROR_10. The mutation score can be calculated as $94/(115 - 18) * 100\% = 96.91\%$.

The above example preliminarily illustrated that our method reduces a large number of mutants ($115 - 36 = 79$ mutants), and the selected mutants (the mutants after reduction) can well maintain the mutation adequacy (with 3.09% loss in mutation score). The next section will conduct a serial of experiments to further evaluate the proposed method.

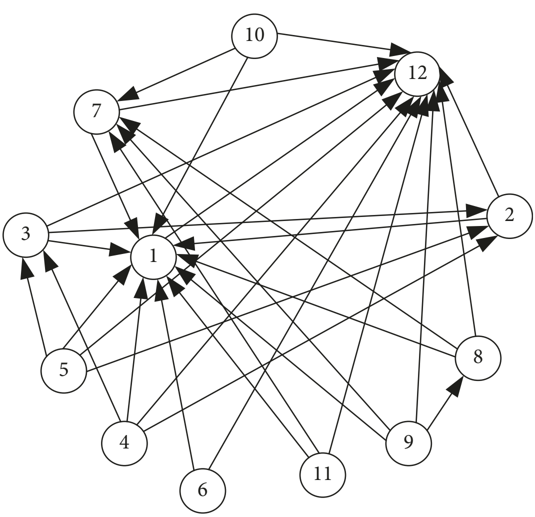


Figure 2: The dominance relation graph of the program in Figure 1.

Table 2: The generated mutants.

s_i	Generated mutants	Total
s_1	AOIS_1, AOIS_2, AOIS_3, AOIS_4, AOIS_5, AOIS_6, AOIS_7, AOIS_8, *ROR_1*, ROR_2, ROR_3, ROR_4, ROR_5, COI_1, LOI_1, LOI_2	16
s_2	AOIS_9, AOIS_10, AOIS_11, AOIS_12, AOIS_13, AOIS_14, AOIS_15, AOIS_16, *ROR_6*, ROR_7, ROR_8, ROR_9, ROR_10, COI_2, LOI_3, LOI_4	16
s_3	AOIS_17, AOIS_18, AOIS_19, AOIS_20, AOIS_21, AOIS_22, AOIS_23, AOIS_24, COI_3, LOI_5, LOI_6	11
s_4	AOIU_1,, AOIS_25, AOIS_26, *AOIS_27*, *AOIS_28*, LOI_7	6
s_5	AOIU_2, AOIS_29, AOIS_30, *AOIS_31*, *AOIS_32*, LOI_8	6
s_6	AOIU_3, AOIS_33, AOIS_34, *AOIS_35*, *AOIS_36*, LOI_9	6
s_7	AOIS_37, AOIS_38, AOIS_39, AOIS_40, AOIS_41, AOIS_42, AOIS_43, AOIS_44, *ROR_11*, ROR_12, ROR_13, ROR_14, ROR_15, COI_4, LOI_10, LOI_11	16
s_8	OIS_45, AOIS_46, AOIS_47, AOIS_48, AOIS_49, AOIS_50, AOIS_51, AOIS_52, *ROR_16*, ROR_17, ROR_18, ROR_19, ROR_20, COI_5, LOI_12, LOI_13	16
s_9	AOIU_4, AOIS_53, AOIS_54, *AOIS_55*, *AOIS_56*, LOI_14	6
s_{10}	AOIU_5, AOIS_57, AOIS_58, *AOIS_59*, *AOIS_60*, LOI_15	6
s_{11}	AOIU_6, AOIS_61, AOIS_62, *AOIS_63*, *AOIS_64*, LOI_16	6
s_{12}	AOIU_7, *AOIS_65*, *AOIS_66*, LOI_17	4

4. Experiment and Analysis

To evaluate our method, this section first puts forward the questions needed to answer; then, the programs are given for experiment, and the experimental process is described; finally, the experimental results are listed and analyzed.

4.1. Research Questions. The aim of identifying the nondominated statements is to select mutants, thus reducing the cost in mutation testing. In view of this, the following questions need to be answered:

(1) Can the proposed method reduce the number of mutants? To answer this question, we first identify the nondominated statements after analyzing dominance relations among statements in the program. Then, the mutants generated from the nondominated statements are selected. Finally, the reduction rate is calculated by the ratio of the

Table 3: Executing the selected mutants on the given test cases.

t_i	Test case	Killed mutants	Total
t_1	assertEquals (9, Mid.getMid (9, 9, 2))	AOIS_53, AOIS_54, AOIU_4, LOI_14	4
t_2	assertEquals (5, Mid.getMid (5, 4, 7))	AOIS_29, AOIS_30, AOIU_2, LOI_8	4
t_3	assertEquals (3, Mid.getMid (2, 3, 8))	AOIS_61, AOIS_62, AOIU_6, LOI_16	4
t_4	assertEquals (−8, Mid.getMid (−4, −8, −8))	AOIS_33, AOIS_34, AOIU_3, LOI_9	4
t_5	assertEquals (−7, Mid.getMid (1, −8, −7))	AOIS_25, AOIS_26, AOIU_1, LOI_7	4
t_6	assertEquals (7, Mid.getMid (8, 7, 6))	—	0
t_7	assertEquals (6, Mid.getMid (7, −6, 6))	—	0
t_8	assertEquals (7, Mid.getMid (7, 1, 7))	—	0
t_9	assertEquals (−1, Mid.getMid (−5, −1, −1))	—	0
t_{10}	assertEquals (6, Mid.getMid (3, 7, 6))	AOIS_57, AOIS_58, AOIU_5, LOI_15	4
t_{11}	assertEquals (5, Mid.getMid (5, 7, 5))	—	0

Table 4: Executing all the mutants against the effective test cases of the selected mutants.

t_i	Test case	Killed mutants	Total
t_1	assertEquals (9, Mid.getMid (9, 9, 2))	AOIS_2, AOIS_3, AOIS_4, AOIS_45, AOIS_46, AOIS_47, AOIS_48, AOIS_53, AOIS_54, AOIS_6, AOIU_4, AOIU_7, COI_5, LOI_12, LOI_14, LOI_17, ROR_17, ROR_18, ROR_19	19
t_2	assertEquals (5, Mid.getMid (5, 4, 7))	AOIS_1, AOIS_21, AOIS_22, AOIS_23, AOIS_24, AOIS_29, AOIS_30, AOIU_2, COI_1, COI_2, COI_3, LOI_1, LOI_3, LOI_5, LOI_8, ROR_2, ROR_3, ROR_4, ROR_7, ROR_8, ROR_9	21
t_3	assertEquals (3, Mid.getMid (2, 3, 8))	AOIS_37, AOIS_38, AOIS_39, AOIS_40, AOIS_5, AOIS_61, AOIS_62, AOIS_7, AOIS_8, AOIU_6, COI_4, LOI_11, LOI_16, LOI_2, ROR_12, ROR_13, ROR_15, ROR_5	18
t_4	assertEquals (-8, Mid.getMid (−4,−8,−8))	AOIS_13, AOIS_15, AOIS_16, AOIS_33, AOIS_34, AOIS_9, AOIU_3, LOI_9	8
t_5	assertEquals (−7, Mid.getMid (1,−8,−7))	AOIS_10, AOIS_11, AOIS_12, AOIS_17, AOIS_18, AOIS_19, AOIS_20, AOIS_25, AOIS_26, AOIU_1, LOI_4, LOI_7	12
t_{10}	assertEquals (6, Mid.getMid (3, 7, 6))	AOIS_41, AOIS_42, AOIS_43, AOIS_44, AOIS_49, AOIS_50, AOIS_51, AOIS_52, AOIS_57, AOIS_58, AOIU_5, LOI_10, LOI_13, LOI_15, ROR_14, ROR_20	16

Table 5: The programs under test.

ID	Program	LOCs	Classes	Methods		Mutants	Description
				Total	Tested		
J1	Trash and take out	30	1	2	2	111	Not reported
J2	Triangle	36	1	1	1	325	Return a triangle type with three integer inputs
J3	Cal	50	1	2	2	315	Calculate the days between two dates in the same year
J4	Math	1233	3	25	5	1506	org.apache.commons.lang3.math
J5	Digest	1038	9	101	24	1401	org.apache.commons.codec.digest
J6	Text	3040	11	320	6	629	org.apache.commons.lang3.text
J7	Cli	2665	23	208	2	264	org.apache.commons.cli
J8	lang3	9344	29	778	37	3494	org.apache.commons.lang3
J9	Linear	12416	67	942	22	5153	org.apache.commons.math3.linear
Total		29852	145	2379	101	13198	

number of reduced mutants to the number of total mutants.

(2) Can the selected mutants well maintain the mutation adequacy? To answer this question, we first execute all the mutants against the given test cases and calculate the mutation score. Then, we execute the selected mutants against the same test cases. Finally, we execute all the mutants against the effective test cases of the selected mutants and investigate the changes in mutation scores so as to analyze whether the selected mutants can well maintain the mutation adequacy.

(3) Is the proposed method more effective than the mutant sampling method? To answer this question, we first

detect the equivalent mutants from the selected mutants. Then, for each program, we randomly sample the same number of nonequivalent mutants according to the number of nonequivalent selected mutants and execute the sampled mutants against the given test cases. Finally, we execute all the mutants against the effective test cases of the sampled mutants and investigate the changes in mutation scores between random sampling method and our method.

4.2. Experimental Programs. Nine programs (packages) in Table 5 are selected for the experiments. Among these programs, J1–J3 are commonly used in mutation testing and the detailed information can be obtained from [20]. J4–J9 are from the open source projects of Apache (http://commons.apache.org), wherein J4 extends java.math and provides business mathematical functionality (http://commons.apache.org/proper/commons-lang/javadocs/api-release/index.html); J5 simplifies common MessageDigest tasks and provides a compatible crypt method that supports MD5, SHA-256, and SHA-512 (http://commons.apache.org/proper/commons-codec/apidocs/index.html); J6 provides classes for handling and manipulating text (http://commons.apache.org/proper/commons-lang/javadocs/api-release/index.html); J7 provides an API for parsing command line options passed to programs (http://commons.apache.org/proper/commons-cli/); J8 provides highly reusable static utility methods, chiefly concerned with adding value to the java.lang classes (http://commons.apache.org/proper/commons-lang/javadocs/api-release/index.html); J9 provides linear algebra support, such as calculating the LUP-decomposition of a square matrix (http://commons.apache.org/proper/commons-math/javadocs/api-3.0/index.html).

The total LOC (lines of code) of the programs in Table 5 is 29852. These programs include 145 classes and 2379 methods, and 101 methods are tested. The methods that are not tested have one of the following features: (1) simple methods like getXXX () or setXXX (), (2) small methods with several or ten lines of code, and (3) the methods that only generate tens or dozens of mutants.

In our experiments, the mutants are generated and executed automatically by using MuClipse. MuClipse is a MuJava plugin for Eclipse, and it provides fifteen traditional mutation operators. After applying all the traditional mutation operators to nine programs, 13198 mutants are generated as listed in Table 5. Among these programs, the largest number of mutants is 5153 from J9 and the least number of mutants is 111 from J1.

4.3. Experimental Process. The experiments are conducted under following environment: Intel(R) Core(TM) i5-4590 CPU @ 3.30 GHz 3.30 GHz, 16.0 GB ROM, Microsoft Windows 7 Service Pack 1 operating system, and Eclipse SDK 4.2.2 with MuClipse 1.3 plugin. MuClipse can provide the state (killed or alive) of each mutant after executing against the given test cases and compare a mutant with the original program in the view of “Mutants and Results.” All the test cases are programmed in the form of JUnit assertions.

For each program in Table 5, we first analyze the dominance relations between the statements. Then, we build the dominance relation graph and search the nondominated statements according to the corresponding vertices with zero in-degree. Finally, the mutants generated from the nondominated statements are selected, and these mutants are the ones after reduction.

Dominance identification is of critical importance in our method. We identify the dominance relations among the statements by manual analysis, since it is very difficult to establish the semantic rules for automatic identification. It seems that the dominance relations among statements can be approximately identified by executing the program against test cases; however, the obtained dominance relations are not semantically accurate so that the selected mutants will be of poor quality. In view of that the precision of identifying dominance relations may be affected by some factors, such as the testers’ skills and familiarity with the programs, we always check the results (the nondominated statements) one or more times by means of walkthrough and inspection.

Test cases are needed to execute the mutants before and after reduction, so as to evaluate the effectiveness of our method in maintaining the mutation adequacy. In our experiments, the test cases of program J1-J3 are genera (Cted by Randoop in Eclipse). Randoop can automatically generate JUnit test cases for Java classes by setting the number of test cases or the time limitation. And the numbers of generated test cases for J1–J3 are listed in Table 6 (Column “Test cases”). The test cases of J4–J9 are provided along with the corresponding open source projects, and the test cases for a method are selected according to the corresponding test class(es) and test method(s). Sometimes, considering that a test method in a test class includes several test cases, we divided the test method into several test methods, and each test method includes a test case.

Although some programs in Table 5 usually appear in previous studies, these programs are coded in various languages or mutated by different mutation tools. As a result, before calculating the mutation scores, the equivalent mutants of all the programs in Table 5 should be detected. And for this, the mutants of each program are executed against the given test cases, and the mutants alive are manually analyzed in the “Compare Mutants” panel. When a mutant alive is semantically different from its original program, the mutant alive is equivalent. And the numbers of detected equivalent mutants of all the programs are listed in Table 6.

4.4. Experimental Results and Analysis

4.4.1. Mutant Reduction. The proposed method is applied to the programs in Table 5, and the mutant reduction rates are listed in Table 7. And Table 6 also lists the numbers of statements and mutants before and after reduction. From Table 7, we have the following observations: (1) the number of mutable statements of all the programs is 1776, and these statements

Table 6: The mutation testing of the mutants before reduction.

ID	Test cases	Mutants	Equivalent mutants	Killed mutants	Mutation score (%)	Effective test cases
J1	7	111	29	82	100	4
J2	300	325	40	276	96.84	28
J3	100	315	43	248	91.18	11
J4	146	1506	178	1160	87.35	76
J5	41	1401	195	1178	97.68	19
J6	104	629	71	476	85.30	49
J7	13	264	32	206	88.79	9
J8	653	3494	287	2908	90.68	265
J9	129	5153	230	4886	99.25	79
Total	1493	13198	1105	11420	Avg. = 93.01	540

Table 7: The mutant reduction.

ID	Before reduction		After reduction		Reduction rate (%)
	Statements	Mutants	Nondominated statements	Mutants	
J1	9	111	2	25	77.48
J2	20	325	6	114	64.92
J3	24	315	8	104	66.98
J4	175	1506	43	361	76.03
J5	221	1401	36	230	83.58
J6	89	629	30	148	76.47
J7	28	264	7	42	84.09
J8	597	3494	157	825	76.39
J9	613	5153	95	971	81.16
Total	1776	13198	384	2820	Avg. = 76.34

generate 13198 mutants; (2) after applying our method to the nine programs, 384 nondominated statements are obtained, and these nondominated ones generate 2820 mutants.

Obviously, the nondominated statements account for only 21.62% (384/1776) of all the mutable statements. Among all the programs, the lowest ratio of the nondominated statements is 15.50% (95/613) from J9, and the highest is 33.71% (30/89) from J6. "Reduction rate" in Table 7, calculated by the ratio of the number of reduced mutants to the number of all the mutants, reflects the effectiveness of the proposed method. After dominance analysis, our method reduces 10378 (13198–2820) mutants and achieves an average reduction rate of 76.34%.

Table 7 suggests that many mutants are reduced, and the mutant reduction rates range from 64.92% (from J2) to 84.09% (from J7). The reason of the high mutant reduction rate is that nearly 80% mutable statements are dominated and the generated mutants are reduced. However, for a program, the number of mutants relates not only to the number of statements but also to other factors such as the variables that the statements have and the mutation operators can be performed. Conclusively, our method can reduce a large number of mutants by only selecting the mutants generated from the nondominated statements.

4.4.2. Effectiveness of the Selected Mutants. In order to check whether the selected mutants (the mutants after reduction) can represent all the mutants to evaluate the given test cases, we conduct the experiments as following steps: (1) We first execute all the mutants against the given test cases to calculate the mutation score as listed in Table 6. (2) The effective test cases are obtained (in Table 8) after executing the selected mutants against the given test cases. (3) All the mutants are executed against the effective test cases obtained in Step (2) to check whether the effective test cases in Table 8 can be effective for all the mutants.

The experiment results are listed in Tables 6 and 8, and from Table 6, we have three observations: (1) all the programs generate 13198 mutants including 1105 equivalent ones (account for 8.37%), and the number of nonequivalent mutants is 12093 (13198 – 1105). (2) The number of the given test cases is 1493. (3) All the test cases kill 11420 mutants, and the average mutation score is 93.01%. Among these programs, the highest mutation score is 100% from J1, and the lowest is 85.30% from J6. "Effective test cases" in Table 6 are the ones that kill mutants. Although the number of given test cases is 1493, the effective ones are only 540. This illustrates that many test cases in the given test cases are redundant.

Table 8 shows the effectiveness of the selected mutants. From Table 8, we have four observations: (1) The number of selected mutants is 2820 including 308 equivalent ones (account for 10.92%), and the number of the nonequivalent mutants after reduction is 2512 (2820 – 308). (2) The given test cases (1493 test cases or 540 effective test cases in Table 6) kill 2336 selected mutants, and the average mutation score is 91.73%. Among these programs, the highest mutation score is 99.12% from J9 and the lowest is 85.48% from J4. (3) The number of effective test cases of the selected mutants is 384, which is 151 (540 – 384) less than the number of effective test cases in Table 6. (4) 384 effective test cases kill 11199

Table 8: The effectiveness of the mutants after reduction.

ID	Selected Mutants	Equivalent Mutants	Killed Mutants	Mutation Score (%)	Effective Test cases	Mutation testing of all the mutants	
						Killed mutants	Mutation score (%)
J1	25	4	18	85.71	3	77	93.90
J2	114	25	84	94.38	19	265	92.98
J3	104	16	84	95.45	9	240	88.24
J4	361	51	265	85.48	62	1139	85.77
J5	230	40	179	94.21	13	1166	96.68
J6	148	32	107	92.24	34	466	83.51
J7	42	10	29	90.63	9	206	88.79
J8	825	73	664	88.30	194	2843	88.65
J9	971	57	906	99.12	194	2843	88.65
Total	2820	308	2336	Avg. = 91.73	384	11199	Avg. = 90.66

mutants after executing all the mutants against the effective test cases of the selected mutants, and the average mutation score is 90.66%.

The results from Tables 6 and 8 suggest that (1) the given test cases achieve 93.01% mutation score after executing all the mutants, as shown in Table 6; (2) and the effective test cases of the selected mutants achieve 90.66% mutation score after executing all the mutants. Therefore, the mutation adequacy drops 2.35% when adopting the selected mutants to evaluating the given test cases instead of all the mutants. The drop of 2.35% in mutation score indicates that a few valuable mutants are omitted by only selecting the mutants generated from the nondominated statements. And compared with the 76.34% mutant reduction rate, the drop in mutation score seems subtle.

It should be noted that our manual tasks may make mistakes when analyzing dominance relations between statements, and the influences of the mistakes are as follows: (1) If the nondominated statements are mistaken as dominated ones, we will obtain a higher reduction rate but a lower mutation adequacy since more mutants (including some valuable mutants) are reduced. (2) If the dominated statements are mistakenly determined as nondominated ones, a lower reduction rate will be obtained without further losing mutation adequacy.

4.4.3. Comparison with Mutant Sampling Method. Mutant sampling is a very simple and practical method for reducing mutants, which randomly samples a percentage of mutants to execute mutation testing, and the sampled mutants are the ones after reduction. Our method and mutant sampling share following similar features: (1) they are for all the traditional mutation operators of MuClipse instead of some mutation operators; (2) they sample (select) a subset of mutants to execute mutation testing. Moreover, our method still has following specific features: (1) we select mutants according to the nondominated statements instead of randomly sampling; (2) the reduction rate of our method cannot be predetermined. Given this, we compare our method with mutant sampling method.

Considering that the randomness of mutant sampling cannot ensure the same number of equivalent mutants in the sampled mutants as our method, we eliminate this influence by only sampling the nonequivalent mutants. Consequently, we first randomly sample the same number of nonequivalent mutants according to Table 8; then the sampled mutants are executed against the given test cases; finally, all the mutants are executed against the effective test case of the sampled mutants. Table 9 lists the results of above experiments.

From Table 9, we have following observations: (1) the same number (2512) of nonequivalent mutants are sampled by the mutant sampling method; (2) 2188 mutants are killed after executing these sampled mutants against the given test cases (1493 test cases), and the average mutation score is 85.24%; (3) the number of effective test cases for the sampled mutants in Table 9 is 342; and (4) these effective test cases kill 10764 mutants, and the mutation score is 85.22%.

The comparison between our method and mutant sampling suggests that (1) the same test cases achieve different mutation scores after executing the sampled (selected) mutants. In detail, the mutation score in Table 9 is 85.24%, which is 6.49% less than that in Table 8. (2) The number of effective test cases in Table 9 is 342, and these test cases achieve 85.22% mutation score after executing all the mutants (5.44% less than that in Table 8, 7.79% less than that in Table 6). This further illustrates that much more valuable mutants have been reduced by the mutant sampling method.

4.4.4. Hypothesis Test of the Experimental Results. We conduct Wilcoxon (Mann–Whitney) test to scientifically reflect whether there is a significant difference between the mutation scores before and after mutant reduction at 0.05 significance level, and the test results are listed in Table 10.

In Table 10, the P value of the Wilcoxon test between the mutation scores after mutant reduction by our method and before mutant reduction is 0.289. It means there is no significant difference since $0.289 > 0.05$, and we mark "–" in the column "Analysis result." However, the P value between the mutation scores after mutant reduction by mutant sampling and before mutant reduction is 0.024 ($0.024 < 0.05$), indicating there is a significant difference (see the mark "+" in Table 10). Therefore, our method can well maintain the mutation adequacy.

From the above results, we can draw conclusions as follows: our method can reduce a large number of mutants without significant loss in the mutation adequacy. However, with the same reduction rate as our method, mutant sampling has a significant loss in the mutation score.

Table 9: The mutant sampling method.

ID	Sampled Mutants	Killed Mutants	Mutation Score (%)	Effective Test cases	Mutation testing of all the mutants	
					Killed mutants	Mutation score (%)
J1	21	16	76.19	3	73	89.02
J2	89	86	96.63	18	205	71.93
J3	88	85	96.59	9	234	86.03
J4	310	244	78.71	54	1101	82.91
J5	190	153	80.53	13	1132	93.86
J6	116	89	76.72	27	436	78.14
J7	32	27	84.38	6	199	85.78
J8	752	620	82.45	179	2702	84.25
J9	914	868	94.47	33	4682	95.23
Total	2512	2188	Avg. = 85.24	342	10764	Avg. = 85.22

Table 10: Wilcoxon test between the mutation scores after and before reduction.

Reduction method	P value	Significance level	Analysis result
Our method	0.289	0.05	−
Random mutant sampling	0.024	0.05	+

5. Conclusion

The high cost in mutation testing resulting from a large number of mutants, which seriously impacts the practical application of mutation testing, can be saved by reducing mutants.

A new method for mutant reduction is proposed by analyzing the dominance relations between statements in the program. And the main contributions of this article are as follows: (1) A dominance based method is proposed for reducing mutants. The proposed method first analyzes the dominance relations between statements; the dominance relation graph is built for aiding to determine the non-dominated statements. Finally, we only select the mutants generated from the nondominated statements for mutation testing. (2) The effectiveness of the proposed method is preliminary illustrated by an example. And after applying the proposed method to nine programs, the experimental results suggest that our method can significantly reduce the number of mutants, and the mutants after reduction can maintain a high mutation adequacy. Additionally, we also compare our method with mutant sampling, and the results suggest that our method is more effective.

In our method, the nondominated statements are manually detected. Manual analysis is quite accurate but time consuming. We will develop tools to effectively analyze the dominance relations and determine the nondominated statements, thus enhancing the practicability of our method. And the redundant ones among mutants before or even after mutant reduction by our method, as pointed out by Papadakis [8], inflate the mutation score; therefore, it is necessary and promising for us to design a trustable mutation tool to generate mutants without redundant ones.

Data Availability

Additionally, our original experimental resources include many programs and excels. These files are sophisticated to be transformed into CIF; therefore, we send the source files as an attachment of the response e-mail.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was jointly supported by the National Natural Science Foundation of China (Nos. 61872168, 61502212, 61573362, 61703188, and 61902161), Research Support Program for Doctorate Teachers of Jiangsu Normal University (Nos. 17XLR042 and 17XLR001), and Guangxi Key Laboratory of Trusted Software (No. kx201704).

Supplementary Materials

The detailed information of our experimental process. (*Supplementary Materials*)

References

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279–290, 1977.

[2] R. A. Demillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] J.-F. Chen, Y.-S. Lu, and X.-D. Xie, "Research on software fault injection testing," *Journal of Software*, vol. 20, no. 6, pp. 1425–1443, 2009.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the International Conference on Software Engineering*, pp. 402–411, Saint Louis, MO, USA, May 2005.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[6] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[7] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information and Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, Amsterdam, Netherlands, 2019.

[9] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 331–341, Beijing, China, July 2012.

[10] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 235–245, San Jose, CA, USA, July 2013.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[12] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: a collateral experiment," in *Proceedings of the Software Engineering Conference*, pp. 300–309, Kerala, India, February 2011.

[13] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the International Conference on Software Engineering*, Hyderabad, India, May-June 2014.

[14] M. Harman, R. Hierons, and S. Danicic, *The Relationship between Program Dependence and Mutation Analysis*, Kluwer Academic Publishers, Dordrecht, Netherlands, 2000.

[15] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[16] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[17] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[18] V. H. S. Durelli, J. Offutt, and M. E. Delamaro, "Toward harnessing highlevel language virtual machines for further speeding up weak mutation testing," in *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 681–690, Montreal, Quebec, Canada, April 2012.

[19] S.-W. Kim, Y.-S. Ma, and Y.-R. Kwon, "Combining weak and strong mutation for a noninterpretive java mutation system," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 647–668, 2013.

[20] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, pp. 691–723, 2011.

[21] J. Shan, Y. Gao, M. liu, J. Liu, L. Zhang, and J. Sun, "A new approach to automated test data generation in mutation testing," *Chinese Journal of Computers*, vol. 31, no. 6, pp. 1025–1034, 2008.

[22] S. Xu, "A method of simplifying complexity of mutation testing," *Journal of Shanghai University*, vol. 13, no. 5, pp. 524–531, 2007.

[23] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems & Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

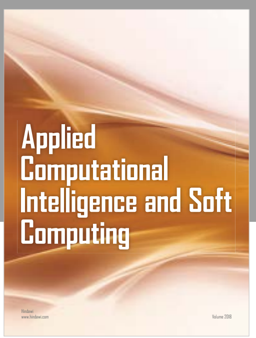
[24] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 21–30, Cleveland, OH, USA, March-April 2014.

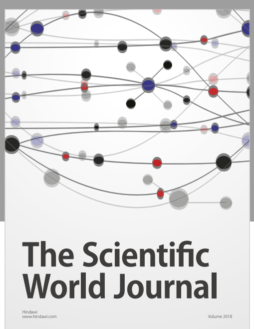
[25] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 176–185, Cleveland, OH, USA, March-April 2014.

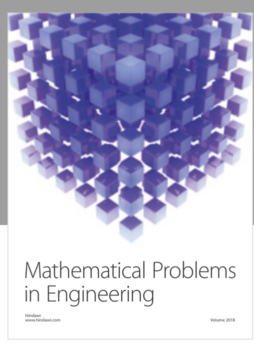
[26] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 1–10, Graz, Austria, April 2015.

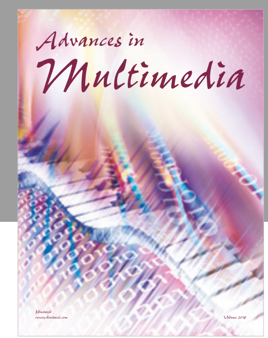
[27] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Information and Software Technology*, vol. 81, pp. 82–96, 2017.

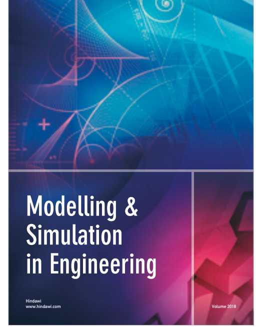
[28] H. Agrawal, "Dominators, super blocks, and program coverage," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 25–34, ACM, Portland, OR, USA, January 1994.











Submit your manuscripts at
www.hindawi.com