Analysis of test suite reduction with enhanced tie-breaking techniques

Jun-Wei Lin, Chin-Yu Huang * 

Department of Computer Science, National Tsing Hua University, No. 101, Section 2, Kuang Fu Road, Hsinchu 300, Taiwan

Abstract

Test suite minimization techniques try to remove redundant test cases of a test suite. However, reducing the size of a test suite might reduce its ability to reveal faults. In this paper, we present a novel approach for test suite reduction that uses an additional testing criterion to break the ties in the minimization process. We integrated the proposed approach with two existing algorithms and conducted experiments for evaluation. The experiment results show that our approach can improve the fault detection effectiveness of reduced suites with a negligible increase in the size of the suites. Besides, under specific conditions, the proposed approach can also accelerate the process of minimization.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

As software develops and evolves, new test cases are continually generated to validate the latest modifications. As a result, the sizes of test suites grow over time. However, a percentage of test cases in a test suite may become redundant, because the requirements executed by one test case may also be validated by others. Due to the constraints on time and resources, it may be impossible to rerun all the test cases whenever the software is modified. Therefore, it is desirable to keep test suite sizes manageable by removing redundant test cases. This process is called test suite minimization (also known as test suite reduction). The problem of finding a minimal size subset from an unminimized test suite to satisfy the same requirements is called the test suite minimization problem [10]. A classical greedy heuristic [4,16] solves this problem by repeating the following two steps: (1) pick the test case which meets the most requirements (random-selection if multiple candidates exist), and (2) remove the requirements covered by the selected test case. They stop when all requirements are satisfied. Another heuristic developed by Harrold et al. [10] selects a representative set of test cases from a test suite, but it may take considerable computing effort in the recursion of the selecting process.

A potential drawback of existing minimization approaches is that they may significantly decrease the fault detecting capability. In the literature, there exist some conflicts among research related to test suite minimization. Wong et al. [17,32] reported that while the all-uses coverage was kept constant, test suites could be minimized at little or no cost to their fault detection effectiveness. Later, Rothermel et al. [6,23] argued that the fault detection capabilities of test suites could be severely compromised by minimization. Intuitively, a test suite which includes more test cases may have a better opportunity to reveal faults. Jeffrey and Gupta [5,20] proposed an approach to test suite reduction which attempts to selectively keep redundant test cases, with the goal of decreasing the loss of fault detection effectiveness. However, the redundancy increases the overhead of maintaining and reusing test suites.

Considering test suite minimization, it may be helpful to pick the test cases that are likely to expose faults instead of including more test cases in the reduced suite. Besides, the computing time of the minimization process may become an issue when the number of test cases and requirements grow. A tie occurs if two or more test cases have the same importance. In this paper, we present a new technique for test suite reduction called reduction with tie-breaking (RTB), which uses additional criterion to break the ties during the minimization process. According to previous studies about the effectiveness of testing coverage criteria [5,28], we believe that our approach can improve the fault-revealing capability of the reduced suites. We integrated our approach with two existing algorithms, and conducted experiments with the Siemens suite programs [28] and the Space program to evaluate and compare the results with prior experimental studies [5,6,23].

The remainder of this paper is organized as follows. In Section 2, we review the test suite reduction problem, the existing solutions, and the empirical studies. The implementation of the proposed approach and the decision process of applying our technique are described in Section 3. Section 4 presents experiments that compare...
existing test suite minimization techniques with our approach. Finally, the conclusion and future work are given in Section 5.

2. Test suite reduction

In this section, we review the definition of the test suite reduction problem, the existing solutions, and the empirical studies for this problem.

2.1. Background and definition

Software is tested with test cases. However, running all of the test cases in a test suite may take a great deal of effort. According to Rothermel et al. [7], one system of about 20,000 lines of code requires seven weeks to run all its test cases. Therefore, eliminating redundant test cases from test suites is desirable. The test suite minimization problem [10] can be formally stated as follows. Given:

\begin{align*}
(1) & \text{ A test suite } T = \{t_1, t_2, t_3, \ldots, t_k\}. \\
(2) & \text{ A set of testing requirements } \{r_1, r_2, r_3, \ldots, r_n\} \text{ that must be satisfied to provide the desired testing coverage of the program.} \\
(3) & \text{ Subsets } \{T_1, T_2, T_3, \ldots, T_n\} \text{ of } T, \text{ one associated with each of the } r_i \text{ s, such that any one of the test cases } t_j \text{ belonging to } T_i \text{ satisfies } r_i.
\end{align*}

Table 1 is an example that shows the coverage information of test cases in a test suite \(\{t_1, t_2, t_3, t_4, t_5\}\). The symbol \(\times\) means satisfaction of a requirement by a test case. Here we find that a subset \(\{t_1, t_2, t_4\}\) of the suite is enough to cover all the requirements \(\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}\), while test cases \(t_2\) and \(t_5\) become redundant since the requirements covered by them are satisfied by the other three test cases. The test suite minimization problem is NP-Complete because the minimum set-covering problem [3,4] can be reduced to this problem in polynomial time. Thus, heuristic solutions for this problem are important.

2.2. The existing solutions

In this subsection, we first introduce the two algorithms that will be modified and evaluated in our experiments. Then we describe other solutions and related works.

<table>
<thead>
<tr>
<th>(r_1)</th>
<th>(r_2)</th>
<th>(r_3)</th>
<th>(r_4)</th>
<th>(r_5)</th>
<th>(r_6)</th>
<th>(r_7)</th>
<th>(r_8)</th>
</tr>
</thead>
</table>
| \(t_1\) | \(\times\) | \(\times\) | \(\times\) | \(\times\)
| \(t_2\) | \(\times\) | \(\times\) | \(\times\) | \(\times\) | \(\times\)
| \(t_3\) | \(\times\) | \(\times\) | \(\times\) | \(\times\)
| \(t_4\) | \(\times\) | \(\times\) | \(\times\) | \(\times\)
| \(t_5\) | \(\times\) | \(\times\) | \(\times\) | \(\times\)

Fig. 1. The HGS algorithm proposed by Harrold, Gupta and Soffa.
This heuristic accepts the associating testing sets $T_i$ for each requirement, and finds a representative set that covers all requirements. It first considers the $T$s of a single element (cardinality one), then places test cases that belong to these $T_i$ into the representative set and marks all $T$s as being selected. Next, the $T_i$ of cardinality two are considered. The test case that occurs the most times among all $T$s of cardinality two is added into the representative set, and all unmarked $T$s containing the test case are marked. This process is repeated until the $T$s of maximum cardinality are examined. When examining the $T$s of cardinality $m$, there may be a tie, because several test cases occur the most among all $T$s of that size. Under the above conditions, the test case that occurs in the most times in a test suite: the essential test cases and the 1-to-1 redundant test cases. A test case is regarded as essential if there exists a requirement which is only covered by the test case. In contrast to the concept of essentials, a test case is said to be 1-to-1 redundant if there exists a test case $t_i$ such that the set of requirements covered by $t_i$ is a subset of the set of requirements covered by $t_j$. For example, Table 2 shows that $r_2$ is 1-to-1 redundant because the requirements of set $[r_1, r_3]$ is the subset of $[r_1, r_2, r_4]$, which is covered by $t_1$. The $HGS$ heuristic algorithm presented by Harrold et al. [10].

2.2.2. The GRE algorithm

Fig. 2 shows the GRE algorithm presented by Chen and Lau [13,15]. In their opinion, there are two special kinds of test cases in a test suite: the essential test cases and the 1-to-1 redundant test cases. A test case is regarded as essential if there exists a requirement which is only covered by the test case. In contrast to the concept of essentials, a test case is said to be 1-to-1 redundant if there exists a test case $t_i$ such that the set of requirements covered by $t_i$ is a subset of the set of requirements covered by $t_j$. For example, Table 2 shows that $r_2$ is 1-to-1 redundant because the requirements of set

### Table 2

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_2$</td>
<td></td>
<td>$x$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x$</td>
<td></td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
</tr>
</tbody>
</table>

2.2.3. Other related works

The classical greedy heuristic for a set-covering problem [4,16] can be applied to the test suite minimization problem. Von Ronne [33] generalized the HGS algorithm, such that every requirement could be satisfied multiple times according to its hitting-factor. Inspiring by the concept analysis framework, Tallam and Gupta [31] developed another heuristic called the delayed-greedy strategy. Concept analysis is a hierarchical clustering technique for objects and their corresponding attributes. When viewing test cases as objects and requirements as attributes, the framework can help expose both the implications among test cases and the implications among those requirements satisfied by the test cases. In their experiments, the delayed-greedy strategy consistently obtained the same or more reduction in suite sizes than it did in prior heuristics, such as in the HGS or in the classical greedy strategy. Black et al. [24] expressed the test suite minimization problem as a binary integer linear programming (ILP) problem. They provided a bi-criteria binary ILP model that considers two objectives simultaneously: minimizing a test suite with regard to a particular level of coverage and maximizing the error detection rates. However, to apply their approach, the prior knowledge of fault detection capability for each test case must be maintained.

Modified condition/decision coverage (MC/DC) is a stricter form of decision (or branch) coverage. To satisfy the criterion for a condition of a decision, a MC/DC pair needs to be covered. By considering the complexity of the criterion, Jones and Harrold [9] described two techniques for test suite reduction: build-up and break-down. The two techniques are tailored for use with MC/DC, and provide a trade-off between effectiveness of reduction and execution time. Genetic algorithms that simulate the mechanism of natural evolution are usually used to find exact or approximate solutions for optimization or searching problems [1]. The algorithms include the concepts of evolutionary biology such as inheritance, mutation and crossover. Based on an integer programming problem formulation and the control flow graphs of programs, Mansour and El-Fakah [11] adapted a hybrid genetic algorithm to the test suite reduction problem. Recently, Zhong et al. [8] presented an experimental study of four typical test suite reduction techniques, including the HGS, the GRE, the genetic-based approach proposed by Mansour and El-Fakah, and the ILP-based approach proposed by Black et al. The main concerns of their study are: (1) execution time, (2) the sizes of the reduced suites, and (3) whether or not the reduced suites produced by different techniques have many test cases in common. They also have provided a
guideline for choosing the appropriate technique. However, they did not address the issue of fault detection capability.

Apart from test suite minimization, another attractive topic is related to test case prioritization. In contrast to the minimization techniques that attempt to remove test cases from a suite, the prioritization techniques [7,19] focus on how to recognize the ordering of a suite for early fault detections. In the late study of Li et al. [18], the effectiveness of several algorithms, including greedy and genetic ones for test case prioritization, were empirically investigated. Because the criteria studied were based on code coverage, their findings could be applied to the test suite reduction problem as well.

2.3. The effect on fault detection capability

A number of empirical studies using existing minimization techniques have been reported. Wong et al. used the ATACMIN tool to minimize the test suites that were not coverage adequate [17,25,32]. Their work shows that when the coverage is kept constant, the size of a test set can be reduced at little or no expense to its fault detection effectiveness. In contrast, the empirical studies conducted in Rothermel et al. [6,23] suggest that reducing test suites can severely compromise the fault detection capabilities of the suites. For test suite filtration and prioritization, Leon and Podgurski [21] compared the coverage-based techniques, such as that of the classical greedy technique with the distribution-based one, which analyzes the distribution of the execution profiles of test cases. The results indicate that both approaches are complementary in the sense that they find different defects. Harder et al. [27] presented a technique for minimizing test suites by considering the operational abstraction. An operational abstraction is a formal mathematical description of the actual behavior of a program. The test case which changed the operational abstraction was retained. The test suites minimized by their technique had better fault detection than the suites reduced by maintaining branch coverage, but were substantially larger.

Coverage criteria (such as branch coverage and all-uses coverage) are used to assess the adequacy of test suites. Some empirical studies on the effectiveness of testing criteria have been performed [12,28,29]. In experiments by Hutchins et al. [28], the tests based on controlflow and dataflow criteria have been frequently complementary in their effectiveness. Recently, Jeffrey and Gupta [5,20] suggested that multiple testing criteria can be used to effectively identify test cases that are likely to expose different faults in software. Their approach (called the RSR technique hereafter) for test suite minimization improves the fault detection effectiveness of the reduced suite, but selectively adds redundancy to the suite.

Sampath et al. [30] presented three strategies, including the tie-breaker concept, for integrating customized usage-based test requirements with traditional test requirements to increase the effectiveness of reduced test suites. However, it should be noted that their approaches are specific to web application testing.

In practice, a software system often contains several hundreds of subprograms. A large number of requirements and test cases may be involved. As a result, the time consumed in the minimization process may become an important issue. To our knowledge, most of prior techniques for improving the fault-revealing capabilities of reduced suites actually increase the sizes of the suites. Although the execution time of different reduction techniques has been compared, the way to accelerate minimization process has not been addressed.

3. RTB: reduction with tie-breaking

In this section, we first provide an example that motivates the proposed RTB approach. Next, we will describe how to integrate RTB with two existing algorithms, HGS and GRE, respectively.

3.1. A motivational example

Fig. 3 shows a simple program and the corresponding branch coverage adequate test suite $T$. This program accepts three integers and returns a value. The branch coverage information of $T$ is shown in Table 3. We first illustrate how to minimize $T$ with the HGS algorithm, namely, by selecting a representative set that covers all the branches of the program. Initially, the branches $B_1^T$ and $B_4^T$ are to be uniquely covered by test cases $t_1$ and $t_4$, respectively. Therefore, we place these two test cases into the representative set, and mark the branches met by them, i.e., $B_1^T$, $B_1^T$, $B_2^T$, $B_4^T$, $B_4^T$, and $B_4^T$. Now, only branch $B_2^T$ is unsatisfied, and test cases $t_3$, $t_5$, and $t_6$ are candidates for covering $B_2^T$. For example, if test case $t_5$ is randomly selected, then the reduced suite generated by the HGS algorithm is $\{t_1, t_2, t_3\}$. It is noticed in this example that test case $t_5$, which exposes a divide-by-zero error at line 15, is eliminated from $T$. In addition, when

### Table 3

<table>
<thead>
<tr>
<th>Branch Coverage Information for $T$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1^T$</td>
</tr>
<tr>
<td>$t_1$</td>
</tr>
<tr>
<td>$t_2$</td>
</tr>
<tr>
<td>$t_3$</td>
</tr>
<tr>
<td>$t_4$</td>
</tr>
<tr>
<td>$t_5$</td>
</tr>
<tr>
<td>$t_6$</td>
</tr>
</tbody>
</table>

Fig. 3. A simple program and the branch coverage adequate suite.
Table 4

The definition-use coverage information for T.

<table>
<thead>
<tr>
<th></th>
<th>(a(1, B_3))</th>
<th>(b(1, B_3))</th>
<th>(c(1, B_3))</th>
<th>(c(1, 15))</th>
<th>(x(3, 9))</th>
<th>(x(5, 9))</th>
<th>(y(7, B_4))</th>
<th>(y(7, 15))</th>
<th>(y(9, B_4))</th>
<th>(y(9, 15))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(t_2)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(t_3)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(t_4)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(t_5)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
</tbody>
</table>

applying the GRE algorithm to \(T\), because \(t_4\), \(t_6\), and \(t_7\) are equally important under the greedy strategy, an arbitrarily chosen from them is still necessary and the test case which reveals a fault, i.e., \(t_5\), may be removed. However, when considering the definition-use pair coverage information of \(T\) shown in Table 4, test case \(t_5\) is more important than \(t_3\) and \(t_4\) since it contains the most definition-use pairs. Therefore, in this example, \(t_5\) is picked from \([t_3, t_4, t_5]\) and the tie is further broken by another coverage criterion. Notice that the reduced suite \([t_1, t_2, t_5]\) exposes the divide-by-zero error at line 15 now.

Fig. 4 provides a decision process that helps project managers determine whether it is suitable to adopt RTB. In software testing, the corresponding data such as test suites, requirements, and coverage information are collected. People may take multiple coverage criteria into consideration when doing software testing. Therefore, when doing test suite reduction, if the selected algorithm makes random choices in the minimization process, and if the other coverage information helps to distinguish test cases, we can then apply RTB to the selected algorithm.

3.2. Implementation

In HGS or traditional greedy algorithms, random selection will be adopted whenever more than one test case has the same importance with respect to the coverage criterion for minimization. However, the random elimination may exclude the test cases which are more likely to detect faults than in the preserved ones. The RSR technique improves the fault detection effectiveness by adding redundancy, which impairs its reducing ability. In fact, the evaluation of software testing efficiency usually takes into account the cardinality of software testing requirements met by a test suite. Therefore, when examining the corresponding data such as test suites, requirements, and coverage information, we will choose two well-known minimization algorithms: the HGS and the GRE, and describe how to integrate our RTB technique with them. The HGS algorithm is common. In many studies, HGS was compared with other algorithms for measuring the performance of test generation [5,6,8,13–15,20,24,26,31], or had served as a basic algorithm for developing new techniques [5,20,26,30,33]. The GRE algorithm is also well-known, and it sometimes performs better than the HGS in minimizing test suites [8,14]. This is the reason for choosing these two algorithms as illustrations.

3.2.1. M-HGS: the modified HGS algorithm by integrating RTB

Fig. 5. The M-HGS; the modified HGS algorithm by integrating RTB. In case of ties, instead of recursive examination or random breakage, our approach immediately selects the test case which covers the most secondary requirements. The algorithm shown in Fig. 5 accepts two inputs: the associating testing sets \(T_s\) for each primary requirement, and \(T'_s\) for each secondary requirement, respectively. In addition, the variable \(curCard\) is used to record the current cardinality under examination, and the \(maxCard\) represents the maximum cardinality among all unmarked \(T_s\). The output of this algorithm is the test suite reduced from \(T\), denoted by \(RS\). It should be noticed that \(RS\) will satisfy all primary testing requirements met by \(T\).

At the outset of the algorithm, the necessary variables will be initialized. After initialization, the algorithm enters a loop which selects the most important test cases and puts them into \(RS\) (initially empty) one-after-the-other (line 22), until all primary requirements are satisfied. In each loop, we take into account the \(T_s\) with cardinality=\(curCard\). Thus, when the algorithm first enters the loop, it finds the \(T_s\) of a single element (cardinality one). Next, it places the test cases that belong to those \(T_s\) into the \(RS\) and marks all \(RS\) covered by the selected test cases. Then the \(T_s\) of cardinality two are considered. The test case that occurs the most times among all \(T_s\) of cardinality two is added into \(RS\) and all unmarked \(RS\) met by the test case are marked. This process will be repeated until the \(T_s\) of the maximum cardinality, i.e., the cardinality=maxCard, are examined.

It is noted that when examining the \(T_s\) of cardinality \(m\), there may be a tie, because several test cases occur the most times among all \(T_s\) of that size. Intuitively, a test case which essentially covers more requirements exercises more elements in a program, and then is likely to expose more faults. Therefore, the function SelectTest regards the total number of secondary requirements (including the marked and the unmarked), covered by each tied test case, as the breaker. If there is still a tie, an arbitrary choice will be returned (line 48). As mentioned in Section 2, the HGS algorithm recursively examines candidates in the minimization process. Thus, in M-HGS, the selection procedure can be simplified and the minimization process is then further accelerated.

3.2.2. M-GRE: the modified GRE algorithm by integrating RTB

The GRE algorithm can be improved by considering the secondary criterion. To adopt the coverage information of secondary crite-
We modify the 1-to-1 redundancy strategy and the greedy strategy as follows:

(a) The modified 1-to-1 redundancy strategy—to remove 1-to-1 redundant test cases. If there are two 1-to-1 redundant test cases, and they satisfy the same set of primary requirements, the test case which covers fewer secondary requirements will be removed. For example, Table 5 shows that test case $t_1$ and $t_2$ are 1-to-1 redundant to each other. We will remove the test case which contributes less coverage with respect to the secondary criterion.

(b) The modified greedy strategy—to select test cases that meet the maximum number of unsatisfied primary requirements. If there is more than one candidate, the test case which satisfies the most secondary requirements will be selected.

Fig. 6 shows the M-GRE, the modified GRE algorithm, by integrating the RTB. It accepts five inputs: the set of primary requirements $R$, the set of all test cases $T$, the associating sets of test cases $T_i$ s for each primary requirement, the associating sets of primary requirements $R_i$ s for each test case, and the associating sets of secondary requirements $R_s$ s for each test case. The output is the reduced suite of $T$, denoted by $Selected$.

First, the essential test cases are picked into $Selected$ (line 18), and the satisfied primary requirements and the test cases in $Selected$ are eliminated from $Unsatisfied_R$ and $Test$ respectively (line 19–20). Next, the algorithm iteratively applies the modified 1-to-1 redundancy strategy and the essential strategy until all primary requirements are met (line 23, 26, and 29). The modified greedy strategy is applied, while both the essential strategy and the modified 1-to-1 redundancy strategy cannot be applied (line 32).

The function $RemoveRedundant$ depicted in Fig. 7 implements the modified 1-to-1 redundancy strategy. The subfunction $Sort$ (line 14) first considers the number of primary requirements satisfied by each test case, i.e. $|R_i|$, and then considers the number of secondary requirements satisfied by each test case with equal $|R_i|$. Thus, when checking 1-to-1 redundancy, those test cases that cover...
more secondary requirements will be preserved with higher chances. Similarly, the function \text{GreedySelect}, shown in Fig. 8, implements the modified greedy strategy. It also calls the \text{Sort} sub-function in the beginning of the function (line 10). This function

```
1 algorithm M-GRE
2 input
3 \( R \): set of all primary requirements, \((r_1, r_2, \ldots, r_k)\)
4 \( T \): set of all test cases, \((t_1, t_2, \ldots, t_n)\)
5 \( T_i \): set of test cases satisfying \( r_i \), \( i = 1, \ldots, n \)
6 \( R_k \): set of primary requirements satisfied by \( t_i \), \( i = 1, \ldots, k \)
7 \( R'_k \): set of secondary requirements satisfied by \( t_i \), \( i = 1, \ldots, k \)
8 output
9 \( \text{Selected} \): resulting set of test cases
10 declare
11 \( \text{Unsatisfied Req} \): set of all unsatisfied primary requirements
12 \( \Delta \text{Selected} \): selected set of test cases in each loop
13 \text{Test} \: set of unselected and non 1-to-1 redundant test cases
14 begin
15 \/* initialization */
16 \( \text{Selected} \: = \) \{\}; \( \text{Unsatisfied Req} \: = \) \( R \); \( \text{Test} \: = \) \( T \).
17 \/* (1) the essential strategy */
18 \( \text{Selected} \: = \) \( \bigcup \{ T_i \mid r_i \in R \text{ and } |T_i| = 1 \} \);
19 \( \text{Unsatisfied Req} \: = \) \( R \: - \) \( \bigcup \{ r_j \mid T_j \in \text{Selected} \} \);
20 \text{Test} \: = \text{Test} \: - \text{Selected}.
21 for-each \( t_i \in (\text{Test} \: - \text{Selected}) \) do \( R_i \: = \) \( R \: - \) \( \bigcup \{ r_j \mid T_j \in \text{Selected} \} \);
22 for-each \( t_i \in \text{Selected} \) do \( R_i \: = \) \{\};
23 while (\( \text{Unsatisfied Req} \: \neq \) \{\}) do
24 \( \Delta \text{Selected} \: = \) \{\};
25 \/* (2) the modified 1-to-1 redundant strategy */
26 \( \text{Unsatisfied Req} \: = \) \( \text{Unsatisfied Req} \: - \) \( \text{RemoveRedundant(Test, R_1, \ldots, R_k, R'_1, \ldots, R'_k, T_1, \ldots, T_n)} \);
27 if (there are some \( T_i \) such that \( r_i \in \text{Unsatisfied Req} \) and \( |T_i| = 1 \)) then
28 \/* (3) the modified greedy strategy */
29 \( \text{Selected} \: = \) \( \bigcup \{ T_i \mid r_i \in \text{Unsatisfied Req} \text{ and } |T_i| = 1 \} \);
30 else
31 \( \text{Selected} \: = \) \( \text{GreedySelect(Test, R_1, \ldots, R_k, R'_1, \ldots, R'_k)} \);
32 end-if
33 \( \text{Test} \: = \) \( \text{Test} \: - \text{Selected} \);
34 \( \text{Test} \: = \) \( \text{Test} \: - \text{Selected} \);
35 \( \text{Unsatisfied Req} \: = \) \( \text{Unsatisfied Req} \: - \) \( \text{RemoveRedundant(Test, R_1, \ldots, R_k, R'_1, \ldots, R'_k, T_1, \ldots, T_n)} \);
36 for-each \( r_i \in \text{Unsatisfied Req} \) do \( T_i \: = \) \( \text{Selected} \);
37 for-each \( r_i \in \text{Selected} \) do \( T_i \: = \) \{\};
38 for-each \( t_i \in \text{Test} \) do \( R_i \: = \) \( \bigcup \{ r_j \mid T_j \in \text{Selected} \} \);
39 for-each \( t_i \in \text{Selected} \) do \( R_i \: = \) \{\};
40 end-while
41 end
42

Fig. 6. M-GRE: the modified GRE algorithm by integrating RTB.

```

```

1 function RemoveRedundant(Test, R_1, \ldots, R_k, R'_1, \ldots, R'_k, T_1, \ldots, T_n)
2 input
3 Test: set of unselected test cases
4 \( R_i \): set of primary requirements satisfied by \( t_i \), \( i = 1, \ldots, k \)
5 \( R'_i \): set of secondary requirements satisfied by \( t_i \), \( i = 1, \ldots, k \)
6 \( T_i \): set of test cases satisfying \( r_i \), \( i = 1, \ldots, n \)
7 output
8 Test: set of unselected and non 1-to-1 redundant test cases
9 \( T_i \): set of non 1-to-1 redundant test cases satisfying \( r_i \), \( i = 1, \ldots, n \)
10 declare
11 \( \text{Redundant} \): set of 1-to-1 redundant test cases
12 size: number of test cases in Test at start
13 begin
14 Sort(Test); \/* sort test cases in Test in descending order of \( |R_i| \) */
15 \/* then sort the test cases with equal \( |R_i| \) in descending order of \( |R'_i| \) */
16 \( \text{Redundant} \: = \) \{\};
17 size \: = \: Test;
18 for \( i = 1 \) to \( \text{size}-1 \) do
19 if (\( t_i \in \text{Test} \)) then
20 for \( j = i+1 \) to \( \text{size} \) do
21 if (\( t_j \in \text{Test} \)) then
22 if (\( R_i \subseteq R_j \)) then \/* check 1-to-1 redundancy */
23 \( \text{Redundant} \: = \) \( \text{Redundant} \: \cup \{ t_j \} \);
24 \( \text{Test} \: = \) \( \text{Test} \: - \{ t_j \} \);
25 end-if
26 end-if
27 end-for
28 end-if
29 end-for
30 for-each \( T_i \) do \( T_i \: = \) \( \text{Redundant} \); \/* remove 1-to-1 redundant test cases */
31 end

Fig. 7. Function RemoveRedundant().
returns that test case which satisfies the most primary requirements. If there is more than one candidate, it returns that test case which contributes the most coverage with respect to the secondary criterion.

4. Experiment and analysis

In Section 3, we described the proposed approaches based on two different algorithms. In this section, we report on the results of the two experiments which were performed in order to evaluate them. First, we compared the M-HGS with the original HGS and the RSR algorithm which evolved from the HGS. Second, we compared M-GRE with the original GRE. In the experiments, the Siemens suite programs [28] and the Space program [22], which were developed in the C language, were used to validate the performance of the proposed approach. Each program was hand-instrumented to record all the coverage information. We implemented all the algorithms in C++.

4.1. Experimental setup

We followed an experimental setup similar to [23]. The subject programs, Siemens programs and the Space program are described in Table 6. All of the programs, faulty versions, and test pools used in our experiments were available from Ref. [34]. We considered branch coverage as the primary testing requirement. To obtain the branch coverage adequate test suites for each program, we first randomly selected varying numbers of test cases from the associated test pool, added them to the suite, and analyzed the branch coverage-based on the selected test cases. If the selected test cases failed to cover all requirements, we added some additional test cases to achieve 100% branch coverage. These additional test cases were randomly selected from the test pool and each of them increases the cumulative branch coverage of the suites. To allow different levels of redundancy, the number of random of test cases we initially added to each suite varied over the sizes ranging from 0 to 0.5 times the number of lines of code in the program. We generated 1000 test suites for each program.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Name} & \textbf{Lines of code} & \textbf{Faulty version size} & \textbf{Test pool size} & \textbf{Description} \\
\hline
tcas & 162 & 41 & 1608 & Altitude separation \\
totinfo & 346 & 23 & 1052 & Information measure \\
schedule & 299 & 9 & 2650 & Priority scheduler \\
schedule2 & 287 & 10 & 2710 & Priority scheduler \\
printtokens & 378 & 7 & 4130 & Lexical analyzer \\
printtokens2 & 366 & 10 & 4115 & Lexical analyzer \\
replace & 514 & 32 & 5542 & Pattern replacement \\
Space & 9127 & 38 & 13,585 & Array definition language interpreter \\
\hline
\end{tabular}
\caption{Siemens suite subject programs.}
\end{table}

In addition, we used the def-use pair coverage as the secondary criterion for the proposed reduction approaches and the RSR technique. The motivation for choosing the def-use pair coverage as the secondary criterion is that the def-use pair coverage is dataflow-based, while the branch coverage is controlflow-based. We have hoped to identify a set of test cases which can exercise different structural and functional elements through these two different kinds of code-based testing criteria, and then improve the fault detection capability of the test suites produced with our approach.

4.2. Measures

In this paper, we use the following criteria to judge the performance of the proposed approach.

- The percentage of suite size reduction (SSR) \cite{5,6,20,23} is defined as
  \[
  \text{SSR} = \frac{|T| - |T_{min}|}{|T|} \times 100\%,
  \]
  where $|T|$ is the number of test cases in the original suite and $|T_{min}|$ is the number of test cases in the minimized/reduced suite. A higher SSR means a better reduction capability.

- The percentage of fault detection effectiveness loss (FDE Loss) \cite{5,6,20,23} is
  \[
  \text{FDE Loss} = \frac{|F| - |F_{min}|}{|F|} \times 100\%.
  \]
  where $|F|$ is the number of distinct faults exposed by the original suite, and $|F_{min}|$ is the number of distinct faults exposed by the minimized/reduced suite. For the subject programs, the fault-exposing information of each test case is provided. Some test cases of a test suite may expose the same faults, but a fault exposed by different test cases of a suite will be counted only once. The closer the FDE Loss is to zero, the better the fault-revealing capability.

- The faults-to-test ratio (FTT ratio) is
  \[
  \text{FTT ratio} = \frac{|F_{min}|}{|T_{min}|}.
  \]
  This is a measure of the number of faults detected by each test case in the reduced suite. This ratio can partially represent the quality (in terms of the fault detection capability) of each test case. A greater faults-to-test ratio means the test cases have better quality on average.

The above three criteria are used to measure the ability and effectiveness of the test case reduction and fault detection. In fact, the time required to finish the reduction process is also an impor-
tient criterion. When we use the HGS algorithm to minimize a test suite, it will invoke at least one recursive function call to break the tie when more than one candidate has equal importance. The recursions may slow down the minimization process and become the bottleneck of the algorithm. Notice that both the M-HGS and RSR evolved from the HGS. Hence, for the first experiment, we counted the occurrences of ties when applying the HGS. The percentage of tie occurrences (PTO) is defined as

$$\text{PTO} = \frac{|C|}{C_{\text{total}}} \times 100\%,$$

where $|C|$ is the number of tie occurrences, and $C_{\text{total}}$ is the total number of candidate selections during minimization. High recursion percentage means that there are a large number of ties during the minimization. In other words, the proposed tie-breaking technique may provide significant improvements in both speed and FDE Loss under the above condition. Besides, we recorded the execution time of minimization programs in both experiments. For the above measures, we computed average values across all 1000 suites for each subject program.

### 4.3. Experimental results of M-HGS

**Suite size reduction:** Table 7 shows the average size of each original test suite, the average size of each reduced test suite, and the average SSR related to all selected approaches. As seen from Table 7, the proposed algorithm (M-HGS) provides almost the same reduction abilities as the HGS. Except for totinfo, the average sizes of reduced suites generated by the RSR were larger than those generated by the HGS and M-HGS; which is because the RSR always selectively keeps redundant test cases with the goal of exposing more faults. Considering totinfo, although the M-HGS gives the lowest value of SSR, the differences compared to HGS and RSR are minor. Overall, compared with HGS, the proposed approach has almost equal ability of reducing test suites for the selected subject programs.

**Fault detection effectiveness loss:** Using (2), we calculated the FDE loss for the three approaches in Table 8. For all the subject programs except for Space, Table 8 clearly shows that the test suites reduced by both M-HGS and RSR can detect more faults than those reduced by the original HGS. Furthermore, compared to RSR, M-HGS also caused less percentage of fault detection effectiveness loss for totinfo, schedule2 and printtokens. Even though the values of FDE loss are not the lowest for other subject programs, M-HGS still has a significant improvement on fault detection effectiveness compared to the original HGS. Although the suites reduced by RSR exposed the most faults for tcas, schedule, printtokens2, and replace, it suffers from the penalty of having the worst SSR. For Space, M-HGS seemed to give the worse performance on FDE Loss when compared to HGS. However, the difference would prove to be not statistically significant in the following paragraph. Table 7 and Table 8 show that, except for Space, when HGS is replaced by M-HGS, M-HGS achieved significant improvements on FDE loss (the maximum reached 14.43%), but it provided almost equal SSR in all subject programs compared with HGS (the deteriorations do not exceed 0.48%).

To determine whether the improvement in fault detection effectiveness we observed for the M-HGS-reduced suites was statistically significant, we conducted a *t-test for matched pairs* [2]. For each of the 1,000 test suites, the number of distinct faults exposed by the HGS-reduced suite and the number of distinct faults exposed by the corresponding M-HGS-reduced suite were considered a matched pair. We assumed that there is no difference in the mean number of faults exposed by the HGS-reduced suites and the M-HGS-reduced suites (the null hypothesis). If the computed *p-value* is less than 0.05 (the *significance level*), statistical practitioners often infer that the null hypothesis is false [2]. The *p*-values computed for our test are shown in Table 9. This indicates that, except for Space, the observed differences are statistically significant. For Space, we do not have strong evidence to reject the null hypothesis.

### Table 7

| Programs | $|T|$ | $|T_{\text{final}}|$ | SSR (%) |
|----------|------|----------------|--------|
| tcas     | 38.83 | 5.00 | 5.12 | 6.77 | 77.12 | 76.87 | 71.96 |
| totinfo  | 82.97 | 5.03 | 5.15 | 5.04 | 86.67 | 86.46 | 86.66 |
| schedule | 71.48 | 3.09 | 3.16 | 5.11 | 90.05 | 89.94 | 85.04 |
| schedule2 | 68.55 | 4.71 | 4.78 | 4.95 | 86.71 | 86.61 | 86.24 |
| printtokens | 91.29 | 6.38 | 6.54 | 7.04 | 87.50 | 87.32 | 82.93 |
| printtokens2 | 87.88 | 5.70 | 5.96 | 8.45 | 86.77 | 86.45 | 82.93 |
| replace | 124.89 | 10.65 | 11.27 | 21.67 | 84.05 | 83.57 | 71.95 |
| Space | 1848.84 | 110.47 | 117.66 | 1825.44 | 82.73 | 82.31 | 1.07 |

### Table 8

| Programs | $|T|$ | $|T_{\text{final}}|$ | FDE Loss (%) |
|----------|------|----------------|-------------|
| tcas     | 17.80 | 6.51 | 6.81 | 8.31 | 56.65 | 55.10 | 47.39 |
| totinfo  | 18.82 | 11.34 | 14.19 | 11.35 | 38.02 | 23.59 | 38.00 |
| schedule | 5.55 | 1.96 | 2.22 | 2.58 | 61.33 | 56.97 | 49.62 |
| schedule2 | 4.67 | 2.07 | 2.37 | 2.10 | 50.28 | 44.62 | 49.58 |
| printtokens | 4.57 | 2.81 | 3.06 | 2.85 | 35.37 | 30.20 | 34.50 |
| printtokens2 | 8.89 | 7.36 | 7.57 | 7.68 | 16.68 | 14.27 | 13.20 |
| replace | 19.07 | 7.66 | 8.61 | 12.40 | 56.80 | 52.18 | 32.62 |
| Space | 33.22 | 27.22 | 27.13 | 33.22 | 15.46 | 16.89 | 0.00 |

1. A *t-test for matched pairs* is a statistical method used to infer the statistical significance of the difference between the means of two populations, given samples where each observation in one sample is logically matched with an observation in the other sample. The testing procedure begins with a *null hypothesis* that assumes the population means are identical, and then computes a *p*-value from the paired data samples. Should the *p*-value be less than a selected *significance level*, the null hypothesis would be rejected.
is clear from Table 11, RSR did not speed up the reduction, while M-HGS saved a very high percentage of execution time (about 84–96% reduction) compared to HGS. This is because the proposed approach breaks the ties by the secondary requirement instead of by recursive examinations. If the proposed approach is adopted in a large-scale software project, the test team can benefit greatly from this characteristic, because the sizes of the test pools and the number of software requirements are considerable. Both RSR and M-HGS can improve the fault detection effectiveness of the reduced test suites compared to HGS. But for most of the subject programs, the sizes of the test suites reduced by M-HGS are less than those reduced by RSR. Further, M-HGS can also save an extremely high percentage of reduction time. On the whole, the proposed M-HGS approach provides a good performance on the Siemens suite programs.

4.4. Experimental results of M-GRE

Suite size reduction: Table 12 shows the average size of each original test suite, the average size of each reduced test suite, and the average SSR related to the two approaches. As seen from Table 12, the proposed algorithm (M-GRE) provides almost equal or better reduction abilities than the GRE. For tcas, totinfo, schedule2, printtokens2 and Space, the average sizes of reduced suites generated by the M-GRE were equal to or smaller than those generated by GRE. Considering the other three subject programs, although the M-GRE gives a lower value of SSR, the differences compared to GRE are extremely minor. Overall, compared with the GRE, the proposed approach has almost the same or better ability of reducing test suites for the selected subject programs.

Fault detection effectiveness loss: Using (2), we calculated the FDE loss for the two approaches in Table 13. The computed p-values for comparing the number of distinct faults exposed by the GRE-reduced suites with the number of distinct faults exposed by the corresponding M-GRE-reduced suites are shown in Table 14. As is clear from Table 13, except for tcas and Space, the test suites reduced by M-GRE

Table 9
Computed p-values of t-test for matched pairs: HGS and M-HGS.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Computed p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>0.0004</td>
</tr>
<tr>
<td>totinfo</td>
<td>0.0000</td>
</tr>
<tr>
<td>schedule</td>
<td>0.0000</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.0000</td>
</tr>
<tr>
<td>printtokens</td>
<td>0.0000</td>
</tr>
<tr>
<td>printtokens2</td>
<td>0.0000</td>
</tr>
<tr>
<td>replace</td>
<td>0.0000</td>
</tr>
<tr>
<td>Space</td>
<td>0.7505</td>
</tr>
</tbody>
</table>

Table 10
Faults-to-test ratio.

<table>
<thead>
<tr>
<th>Programs</th>
<th>HGS</th>
<th>M-HGS</th>
<th>RSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>1.30</td>
<td>1.33</td>
<td>1.23</td>
</tr>
<tr>
<td>totinfo</td>
<td>2.26</td>
<td>2.77</td>
<td>2.25</td>
</tr>
<tr>
<td>schedule</td>
<td>0.65</td>
<td>0.71</td>
<td>0.52</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.45</td>
<td>0.51</td>
<td>0.44</td>
</tr>
<tr>
<td>printtokens</td>
<td>0.46</td>
<td>0.49</td>
<td>0.43</td>
</tr>
<tr>
<td>printtokens2</td>
<td>1.36</td>
<td>1.34</td>
<td>0.95</td>
</tr>
<tr>
<td>replace</td>
<td>0.72</td>
<td>0.77</td>
<td>0.58</td>
</tr>
<tr>
<td>Space</td>
<td>0.25</td>
<td>0.23</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 11
Variations on execution time compared with HGS.

<table>
<thead>
<tr>
<th>Programs</th>
<th>M-HGS</th>
<th>RSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>-85.17%</td>
<td>+1.53%</td>
</tr>
<tr>
<td>totinfo</td>
<td>-94.74%</td>
<td>+0.25%</td>
</tr>
<tr>
<td>schedule</td>
<td>-94.52%</td>
<td>+1.62%</td>
</tr>
<tr>
<td>schedule2</td>
<td>-96.00%</td>
<td>+0.66%</td>
</tr>
<tr>
<td>printtokens</td>
<td>-85.46%</td>
<td>+1.12%</td>
</tr>
<tr>
<td>printtokens2</td>
<td>-84.01%</td>
<td>+0.55%</td>
</tr>
<tr>
<td>replace</td>
<td>-92.41%</td>
<td>+0.50%</td>
</tr>
<tr>
<td>Space</td>
<td>-96.27%</td>
<td>+0.01%</td>
</tr>
</tbody>
</table>

is clear from Table 11, RSR did not speed up the reduction, while M-HGS saved a very high percentage of execution time (about 84–96% reduction) compared to HGS. This is because the proposed approach breaks the ties by the secondary requirement instead of by recursive examinations. If the proposed approach is adopted in a large-scale software project, the test team can benefit greatly from this characteristic, because the sizes of the test pools and the number of software requirements are considerable. Both RSR and M-HGS can improve the fault detection effectiveness of the reduced test suites compared to HGS. But for most of the subject programs, the sizes of the test suites reduced by M-HGS are less than those reduced by RSR. Further, M-HGS can also save an extremely high percentage of reduction time. On the whole, the proposed M-HGS approach provides a good performance on the Siemens suite programs.

4.4. Experimental results of M-GRE

Suite size reduction: Table 12 shows the average size of each original test suite, the average size of each reduced test suite, and the average SSR related to the two approaches. As seen from Table 12, the proposed algorithm (M-GRE) provides almost equal or better reduction abilities than the GRE. For tcas, totinfo, schedule2, printtokens2 and Space, the average sizes of reduced suites generated by the M-GRE were equal to or smaller than those generated by GRE. Considering the other three subject programs, although the M-GRE gives a lower value of SSR, the differences compared to GRE are extremely minor. Overall, compared with the GRE, the proposed approach has almost the same or better ability of reducing test suites for the selected subject programs. Fault detection effectiveness loss: Using (2), we calculated the FDE loss for the two approaches in Table 13. The computed p-values for comparing the number of distinct faults exposed by the GRE-reduced suites with the number of distinct faults exposed by the corresponding M-GRE-reduced suites are shown in Table 14. As is clear from Table 13, except for tcas and Space, the test suites reduced by M-GRE

Table 12
Experiment results for average percentage of suite size reduction.

| Programs | | |
|----------|----------|----------|----------|----------|
| tcas     | 38.86    | 5.10     | 5.05     | 86.88    | 87.00    |
| totinfo  | 81.23    | 5.07     | 5.07     | 93.76    | 93.76    |
| schedule | 70.89    | 3.16     | 3.18     | 95.54    | 95.51    |
| schedule2| 68.16    | 4.82     | 4.82     | 92.93    | 92.93    |
| printtokens| 91.20    | 6.62     | 6.64     | 92.74    | 92.72    |
| printtokens2| 89.12    | 5.78     | 5.76     | 93.51    | 93.54    |
| replace  | 123.17   | 11.49    | 11.57    | 90.67    | 90.61    |
| Space    | 2197.90  | 113.83   | 113.64   | 88.66    | 88.67    |

Table 13
Experiment results for average percentage of fault detection effectiveness loss.

| Programs | | |
|----------|----------|----------|----------|----------|
| tcas     | 17.70    | 6.41     | 6.38     | 63.79    | 63.95    |
| totinfo  | 18.80    | 13.56    | 13.80    | 27.87    | 26.60    |
| schedule | 5.60     | 2.12     | 2.21     | 62.20    | 60.47    |
| schedule2| 4.63     | 2.20     | 2.28     | 52.50    | 50.77    |
| printtokens| 4.57     | 2.98     | 3.08     | 34.83    | 32.68    |
| printtokens2| 8.94     | 7.56     | 7.62     | 15.45    | 14.68    |
| replace  | 19.02    | 8.11     | 8.37     | 57.36    | 55.98    |
| Space    | 33.79    | 27.00    | 26.97    | 19.85    | 19.94    |

Table 14
Computed p-values of t-test for matched pairs: GRE and M-GRE.

<table>
<thead>
<tr>
<th>Program name</th>
<th>Computed p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>0.6361</td>
</tr>
<tr>
<td>totinfo</td>
<td>0.0000</td>
</tr>
<tr>
<td>schedule</td>
<td>0.0000</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.0000</td>
</tr>
<tr>
<td>printtokens</td>
<td>0.0000</td>
</tr>
<tr>
<td>printtokens2</td>
<td>0.0000</td>
</tr>
<tr>
<td>replace</td>
<td>0.0000</td>
</tr>
<tr>
<td>Space</td>
<td>0.4359</td>
</tr>
</tbody>
</table>
can detect more faults than those reduced by the original GRE in all subject programs. Although the suites reduced by GRE exposed more faults for tcas and Space, it suffers from the penalty of having worse SSR. Besides, the differences between the FDE losses of GRE and M-GRE for these two programs have not proven to be statistically significant. Tables 12 and 13 show that for most of the subject programs, compared to GRE, M-GRE provided equal SSR, and on average achieved slight improvements on FDE loss. For tcas and Space, we do not have strong evidence to reject the null hypothesis. For the other programs, the differences between the two methods shown in Table 13 were statistically significant.

Some may argue that the improvement on FDE loss is minor in this experiment. Compared to the original GRE, the proposed approach further deals with the following two conditions: (1) under the 1-to-1 redundancy strategy, there are two 1-to-1 redundant test cases and they cover the same set of primary requirements, and (2) under the greedy strategy, there is more than one test case that satisfies the equal number of primary requirements. According to our observation, the above two conditions seldom happened in the experiment. This may be the reason that our technique cannot get a significant improvement on the FDE Loss.

Faults-to-test ratio: Table 15 shows the average FTT ratio with respect to each program. From Table 15, we can find that the M-GRE gives a good performance, since the values of FTT for all subject programs are the highest. This indicates that the test cases selected by the proposed technique are likely to expose more faults; i.e., the suites reduced by the proposed approach may have a better quality.

Acceleration of minimization process: Table 16 shows the variations on the execution time taken to reduce the test suites when the GRE was replaced by the M-GRE. In our experiments, in general, the M-GRE neither speeded up nor decelerated the reduction. For the M-GRE, we modified and extended the two strategies of GRE and did not deal with the speed issue. Thus, the results are predictable.

The experimental results show that for most of the subject programs, by integrating the proposed RTB technique with the GRE algorithm, we can slightly improve the fault detection effectiveness of reduced test suites while hardly affecting the sizes of the suites. As a result, the FTT ratios of reduced suites are as good as, or better than, those of the GRE-reduced suites.

5. Conclusion and future work

Traditional test suite reduction techniques usually adopt random selection whenever ties occur. Nevertheless, random elimination may exclude the test cases which are more likely to detect faults than the preserved one. In fact, the evaluation of software testing efficiency usually takes into account more than one criterion. Therefore, in this paper, we proposed a new approach, i.e., reduction with tie-breaking (RTB), to enhance the existing techniques. In the proposed framework, an additional coverage criterion was used to break ties during minimization process. To illustrate the concept of RTB, we chose the HGS and the GRE approach, and developed new algorithms. In fact, all existing test suite minimization techniques involving random selection could be integrated into this framework through the proposed decision process.

In the experimental study, the Siemens suite programs and the Space program are used to judge the performance of the proposed approach. In the first experiment, for most of the subject programs, our technique improved the fault detection effectiveness with a negligible increase in the sizes of the reduced suites, and greatly accelerated the minimization process. In the second experiment, the improved fault detection effectiveness of the suites reduced by our technique was not considerable, but the differences were statistically significant for most of the subject programs. Besides, the percentage of suite size reduction produced by our approach is still comparable. As a result, the average number of faults revealed by each test case is raised. The results may mean the proposed approach can refine the selection of test cases. Future research will continue to assess the effectiveness of incorporating other test suite reduction approaches into the proposed framework.

References


