# Test Suite Reduction Analysis with Enhanced Tie-Breaking Techniques

Jun-Wei Lin, Chin-Yu Huang, and Chu-Ti Lin
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
{castman@selab.cs.nthu.edu.tw|cyhuang@cs.nthu.edu.tw|d924390@oz.nthu.edu.tw}

*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. Most of prior works which address this problem affect some extent of suite size reduction. In this paper, we present a novel approach for test suite reduction that uses additional testing criterion to break the ties in the minimization process. We performed an experiment with the Siemens suite subject programs. The experiment results show that, compared to existing approaches, the proposed approach can improve the fault detection effectiveness of reduced suites with negligible increase in the size of the suites. Besides, the proposed approach can also accelerate the process of minimization.**

*Keywords* – **Software testing, testing criteria, test suite reduction, test suite minimization, fault detection effectiveness, tie-breaking.**

## I. INTRODUCTION

As software system 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 time and resource constraints, 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 *test suite minimization problem* [1]. A classical greedy heuristic [2] solves this problem by repeating the following two steps: (1) to pick the test case which meets the most requirements (random selecting if multiple candidates exist); (2) to remove the requirements covered by the selected test case. They stop when all requirements are satisfied. Another heuristic developed by Harrold et al. [1] 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 some test case removals in a test suite may significantly decrease the fault-detecting capability. In the literatures, there exist some conflicts among researches related to test suite minimization. Wong et al. [3] 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. [4] argued that the fault detection capabilities of test suites can be severely compromised by minimization. Intuitively, a test suite which includes more test cases may have a better opportunity to reveal more faults. Jeffrey and Gupta [5] 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, instead of including more test cases in the reduced suite, it may be helpful to pick the test cases that are likely to expose faults. Besides, the computing time of minimization process may become an issue when the amounts of test cases and requirements grow. If there is more than one test case with equal importance to a suite, the elimination of the redundant test cases is in the condition of tie. In this paper, we will 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. According to previous studies about the effectiveness of testing coverage criteria [5], [6], we believe that our approach can improve the fault-revealing capability of the reduced suites. We will also implement our approach and conduct experiments with Siemens suite programs [6] to evaluate and compare the results with prior experimental studies [4], [5].

The remainder of this paper is organized as follows. In section II, we discuss the related work. Our algorithm for test suite reduction is described in section III. Section IV presents experiments that compare existing test suite minimization techniques with our approach. Finally, the conclusion and future work are given in section V.

## II. RELATED WORK

The test suite minimization problem [1] can be formally stated as follows. Given:

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

978-1-4244-2330-9/08/$25.00 ©2008 IEEE

Problem: Find a *minimal cardinality* subset of $T$ that is capable of exercising all $r_i$s exercised by the unminimized test suite $T$.

This problem is equivalent to the NP-Complete *minimum set-covering problem* [7]. Thus, heuristic solutions for this problem are important. Till now, several heuristics for test suite minimization have been proposed. Chvatal [2] proposed a simple heuristic for set-covering problem, which can be applied to the test suite minimization problem. Harrold et al. [1] developed an algorithm called the HGS algorithm, this heuristic accepts the associating testing sets $T_i$s for each requirement respectively, and finds a representative set that covers all requirements. Later, Chen and Lau [8] also proposed the GRE algorithm that uses two dividing strategies to optimally minimize a test suite. Nevertheless, these two strategies cannot be applied to every suite. In recent years, Tallam and Gupta [9] presented another heuristic called *Delayed-Greedy* that exploits both the implications among the test cases and the implications among the requirements to enable further reduction of test suites. Considering the *Modified Condition/Decision Coverage* (MC/DC) criterion, Jones and Harrold [10] described two techniques for test suite minimization. Black et al. [11] proposed a *bi-criteria approach* for test suite minimization that meets the following two objectives simultaneously: minimizing a test suite with regard to a particular level of coverage while maximizing error detection rates. 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 [12], [13] focus on how to recognize an ordering of a suite for early fault detections.

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 [3], [14]. 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 [4] suggest that reducing test suites can severely compromise the fault detection capabilities of the suites. Leon and Podgurski [15] presented an empirical comparison of the coverage-based and distribution-based techniques for test suite minimization and prioritization. The results indicate that both approaches are complementary in the sense that they find different defects.

Testing 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 [6], [16], [17]. In experiments by Hutchins et al. [6], the tests based on controlflow and dataflow criteria are frequently complementary in their effectiveness. Recently, Jeffrey and Gupta [5] 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. [18] 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. Unfortunately, when determining the test cases with better fault-revealing capability, most of the existing techniques for test suite minimization do not avoid affecting the extent of test suite size reduction. Moreover, the acceleration of minimization process is also ignored.

## III. TEST SUITE REDUCTION WITH TIE-BREAKING

In the traditional greedy algorithms or the HGS, the 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 the preserved one. The RSR 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 more than one criterion. Thus, in addition to the *primary* criterion, the importance of these candidates could be further evaluated by another coverage criterion, denoted by the *secondary* criterion. We can break the ties in minimization process by selecting the test case which contributes the most coverage with respect to the secondary criterion. That is, the fault detection effectiveness can be enhanced by using a refined way to select a test case rather than adding redundancies. Consequently the proposed framework still tries to remove all redundancies in a test suite, and then retains the reducing ability.

All test suite minimization techniques involving random selection can be integrated into this framework. Specifically for the HGS algorithm which recursively examines candidates in minimization process, we incorporate the HGS algorithm into this framework to simplify the selection procedure and further accelerate the whole process. In case of ties, instead of recursive examination or randomly breaking, our approach immediately selects the test case which covers the most secondary requirements. The algorithm is developed and depicted in Fig. 1. It accepts two inputs: the associating testing sets $T_i$s for each primary requirement and $T_i^s$s for each secondary requirement, respectively. Besides, the variable *curCard* is used to record the current cardinality under examination, and *maxCard* represents the maximum cardinality among all unmarked $T_i$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 case and puts it into *RS* (initially empty) one-after-the-other (Line 22) until all primary requirements are satisfied. In each loop, we take into account the $T_i$s with cardinality=*curCard*. Thus, when the algorithm first enters the loop, it finds out the $T_i$s of single element (cardinality one). Next, it places test cases that belong to those $T_i$s into the *RS* and marks all $r_i$s covered by the selected test cases. Then the $T_i$s of cardinality two are considered. The test case that occurs the most times among all $T_i$s of cardinality two is added into *RS* and all unmarked $r_i$s met by the test case are marked. This process will be repeated until the $T_i$s of the maximum cardinality, i.e., the cardinality=*maxCard*, are examined.

It is noted that, when examining the $T_i$s of cardinality *m*, there may be a tie because several test cases occur the most times among all $T_i$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.

## IV. EXPERIMENT & ANALYSIS

In the experiments, the Siemens suite programs [6] which were developed in C language were used to validate the performance of the proposed approach. Each program was hand-instrumented to record all the coverage information of the suite. In addition to the proposed approach, the HGS algorithm and the RSR technique were also selected for the purpose of comparisons. We implemented the three approaches in C++.

### A. *Experiment Setup*

We followed an experimental setup similar to [4]. The subject programs, Siemens programs, were described in Table I. All programs, faulty versions, and test pools used in our experiments were available from [19]. We considered the branch coverage as the primary testing

```
1  algorithm ReduceWithTieBrk
2
3     t1, t2, ..., tk: test cases in original (unreduced) test suite T
4     r1, r2, ..., rn: set of primary testing requirements
5     r1^s, r2^s, ..., rm^s: set of secondary testing requirements
6
7  input
8     T1, T2, ..., Tn: subsets of T which meet the primary requirements r1, r2, ..., rn respectively
9     T1^s, T2^s, ..., Tm^s: subsets of T which meet the secondary requirements r1^s, r2^s, ..., rm^s respectively
10 output
11    RS: a reduced subset of T
12
13 begin
14    maxCard ← the maximum cardinality among all Ti's;
15    curCard ← 0;
16    RS ← {};
17    loop
18          curCard ← curCard + 1;
19          while there is at least a Ti of curCard s.t. ri is unmarked do
20                list ← all tests in Ti's of curCard s.t. ri is unmarked;
21                nextTest ← SelectTest(curCard , list);
22                RS ← RS ∪ { nextTest };
23                mayReduce ← false;
24                for each Ti containing nextTest do
25                      mark ri;
26                      if Ti 's cardinality = maxCard then
27                            mayReduce ← true;
28                endfor
29                if mayReduce then
30                      maxCard ← the maximum cardinality among all Ti s.t. ri is unmarked
31          endwhile
32    until curCard = maxCard
33 end
34
35 function SelectTest(size, list)
36    for each test case t in list do
37          count[t] ← number of unmarked Ti's of size containing t;
38    testList ← test cases in list s.t. count[t] is maximum;
39    if the cardinality of testList =1 then
40          return the test case in testList;
41    else                                                  /* there are more than one candidates.*/
42          for each test case t in testList do             /* select the test case which covers more
43                count[t] ← number of Ti^s's containing t;   secondary requirements. */
44          secTestList ← test cases in testList s.t. count[t] is maximum;
45          if the cardinality of secTestList =1 then
46                return the test case in secTestList;
47          else                                            /* ties are broken arbitrarily*/
48                return any test case in secTestList at random;
49 end SelectTest
```

Fig. 1. The implementation of reduction with tie-breaking.

TABLE I
SIEMENS SUITES SUBJECT PROGRAMS

| Name | Lines of Code | Faulty Version Count | Test Pool Size | Description |
|---|---|---|---|---|
| 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 |

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 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. To allow different levels of redundancy, the random amount of test cases we initially added to each suite varied over sizes ranging from 0 to 0.5 times the number of lines of code in the program. We totally generated 1000 test suites for each program.

Besides, we used the def-use pair coverage as the secondary criterion for the proposed reduction approach 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 hope 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 reduced with our approach.

B. *Measures*

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

- The *percentage of suite size reduction* (SSR) [4], [5] is defined as

$$\text{SSR} = \frac{|T| - |T_{\min}|}{|T|} \times 100\%, \quad (1)$$

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. Higher SSR means better reduction capability.

- The *percentage of fault detection effectiveness loss* (FDE Loss) [4], [5] is

$$\text{FDE Loss} = \frac{|F| - |F_{\min}|}{|F|} \times 100\%, \quad (2)$$

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. The closer FDE Loss is to zero, the better the fault reveal capability.

- The *faults-to-test ratio* (FTT ratio) is

$$\text{FTT ratio} = \frac{|F_{\min}|}{|T_{\min}|}. \quad (3)$$

It 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. Greater faults-to-test ratio means the test cases have better quality on average.

The above three criteria are used to measure the ability of the test case reduction and fault detection effectiveness. In fact, the time required to finish the reduction process is also an important 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 RTB and RSR evolved from HGS. Hence, in addition to the consumed time, we also counted the occurrences of ties when applying the HGS. The *percentage of tie occurrences* (PTO) is defined as:

$$\text{PTO} = \frac{|C|}{|C_{total}|} \times 100\%, \quad (4)$$

where $|C|$ is the number of tie occurrences, and $|C_{total}|$ 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 have significant improvement in both speed and FDE Loss under the above condition. For the above measures, we computed average values across all 1000 suites for each subject program.

C. *Discussions*

*Suite Size Reduction:* Table II 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 II, the proposed algorithm (RTB) provides almost the same reduction abilities as the HGS. Except for `totinfo`, the average sizes of reduced suites generated by RSR were larger than those generated by HGS and RTB; that is because the RSR always selectively keeps redundant test cases with the goal of exposing more faults. Considering `totinfo`, although the RTB 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 III. Table III clearly shows that the test suites reduced by both RTB and RSR can detect more faults than those reduced by the original HGS in all subject programs. Further, compared to RSR, RTB also caused less percentage of fault detection effectiveness loss for `totinfo`, `schedule2` and `printtokens`. Even though the values of FDE Loss are not lowest for other subject programs, RTB 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. Table IV lists the deterioration of SSR and the improvement of FDE Loss when HGS is replaced by the proposed RTB. As seen from Table IV, RTB 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%).

*Faults-to-Test Ratio:* Table V shows the average FTT ratio with respect to each Siemens suite program. From Table V, we can find that RTB gives a fairly outstanding performance since the values of FTT for most subject programs are all the highest. Although RTB gives a lower FTT value in `printtokens2` as compared to HGS, the difference is not significant. 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 better quality.

*Acceleration of Minimization Process:* Fig. 2 illustrates the values of PTO for each subject programs, and Table VI shows the variations on the execution time taken to reduce the test suites when HGS was replaced by RTB and RSR, respectively. From Fig. 2, it is found that the values of PTO range between 53% and 83%. That is, the ties frequently occur during the reduction for each program. As is clear from Table VI, RSR did not speed up the reduction while RTB 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 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 test pools and the number of software requirements are considerable.

Both RSR and RTB can improve the fault detection effectiveness of the reduced test suites compared to HGS. But the sizes of test suites reduced by RTB are less than those reduced by RSR. Further, RTB can also save an extremely high percentage of reduction time. On the whole, the proposed approach RTB provides a good performance on the Siemens suite programs.

TABLE II
EXPERIMENT RESULTS FOR AVERAGE PERCENTAGE OF SUITE SIZE REDUCTION

| Programs | \|T\| | \|Tmin\| | | | SSR (%) | | |
|---|---|---|---|---|---|---|---|
| | | HGS | RTB | RSR | HGS | RTB | RSR |
| 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 | 86.45 |
| 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 |

TABLE III
EXPERIMENT RESULTS FOR AVERAGE PERSENTAGE OF FAULT DETECTION LOSS

| Programs | \|F\| | \|Fmin\| | | | FDE Loss (%) | | |
|---|---|---|---|---|---|---|---|
| | | HGS | RTB | RSR | HGS | RTB | RSR |
| tcas | 17.80 | 6.51 | 6.81 | 8.31 | 56.65 | 55.10 | 47.39 |
| totinfo | 18.82 | 11.35 | 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 |

TABLE IV
THE EFFECTS ON SSR AND FDE LOSS WHEN HGS IS REPLACED BY RTB

| Programs | SSR Deterioration | FDE Loss Improvement |
|---|---|---|
| tcas | +0.25 % | +1.55 % |
| totinfo | +0.21 % | +14.43 % |
| schedule | +0.11 % | +4.36 % |
| schedule2 | +0.10 % | +5.66 % |
| printtokens | +0.19 % | +5.16 % |
| printtokens2 | +0.32 % | +2.41 % |
| replace | +0.48 % | +4.62 % |

TABLE V
FAULTS-TO-TEST RATIO

| Programs | HGS | RTB | RSR |
|---|---|---|---|
| tcas | 1.30 | 1.33 | 1.23 |
| totinfo | 2.26 | 2.77 | 2.25 |
| schedule | 0.65 | 0.71 | 0.52 |
| schedule2 | 0.45 | 0.51 | 0.44 |
| printtokens | 0.46 | 0.49 | 0.43 |
| printtokens2 | 1.36 | 1.34 | 0.95 |
| replace | 0.72 | 0.77 | 0.58 |

TABLE VI
VARIATIONS ON EXECUTION TIME COMPARED WITH HGS

| Programs | RTB | RSR |
|---|---|---|
| tcas | −85.17% | +1.53% |
| totinfo | −94.74% | +0.25% |
| schedule | −94.52% | +1.62% |
| schedule2 | −96.00% | +0.66% |
| printtokens | −85.46% | +1.12% |
| printtokens2 | −84.01% | +0.55% |
| replace | −92.41% | +0.50% |

## V. CONCLUSION AND FUTURE WORK

Traditional test suite reduction techniques usually adopt the random selection whenever the ties occur. Nevertheless, the 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

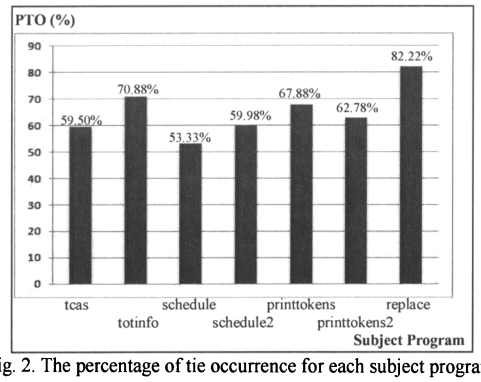


Fig. 2. The percentage of tie occurrence for each subject program.

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 approach as an illustration, and developed a new algorithm. In fact, all existing test suite minimization techniques involving random selection can be integrated into this framework. For example, the GRE heuristic can also be ameliorated by considering the secondary requirement.

In the experimental study, the Siemens suite programs are used to judge the performance of the proposed approach. The experimental results show that it can significantly improve the fault detection effectiveness while the increments on the sizes of test suites are extremely minor. As a result, the average number of faults revealed by each test case is raised. This may mean the proposed approach can refine the selection of test cases. Furthermore, this approach can also greatly shorten the time required to finish the reduction. Finally, future research will continue to assess the effectiveness of incorporating other test suite reduction approaches into the proposed framework.

## ACKNOWLEDGMENT

This work was supported by the National Science Council, Taiwan, under Grant NSC 96-2221-E-007-079.

## REFERENCES

[1] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Transactions on Software Engineering and Methodology,* vol. 2, no. 3, pp. 270–285, Jul. 1993.

[2] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," *Math. Operations Research*, vol. 4, no. 3, pp. 233-235, Aug. 1979.

[3] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software—Practice and Experience*, vol. 28, no. 4, pp. 347-369, Apr. 1998.

[4] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proceedings of the 14th International Conference on Software Maintenance,* pp. 34-43, Nov. 1998, Bethesda, MD, USA.

[5] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 108-123, Feb. 2007

[6] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proceedings of the 16th International Conference on Software Engineering,* pp. 191–200, May 1994, Sorrento, Italy.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[8] T. Y. Chen and M. F. Lau, "A New Heuristic for Test Suite Reduction," *Information and Software Technology*, vol. 40, no. 5-6, pp. 347-354, Jul. 1998.

[9] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *Proceedings of the 6th Workshop Program Analysis for Software Tools and Engineering*, pp. 35-42, Sep. 2005, Lisbon, Portugal.

[10] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195-209, Mar. 2003.

[11] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction," *Proceedings of the 26th International Conference on Software Engineering,* pp. 106-115, May 2004, Scotland, UK.

[12] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929-948, Oct. 2001

[13] A. Srivastava and J. Thiagrajan, "Effectively Prioritizing Tests in Development Environment," *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 97-106, Jul. 2002, Rome, Italy.

[14] J. R. Horgan and S. A. London, "ATAC: A Data Flow Coverage Testing Tool for C," *Proceedings of the 2nd Symposium on Assessment of Quality Software Development Tools*, pp. 2-10, May 1992, LA, USA.

[15] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 442-456, Nov. 2003, Denver, Colorado, USA.

[16] P. Frankl and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, Aug. 1993.

[17] P. G. Frankl and O. Iakounenko, "Further Empirical Studies of Test Effectiveness," *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 153-162, Nov. 1998, Orlando, FL, USA.

[18] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, "Integrating Customized Test Requirements with Traditional Requirements in Web Application Testing," *Proceedings of the Workshop on Testing, Analysis, and Verification of Web Services and Applications,* pp. 23-32, Jul. 2006, Portland, Maine, USA.

[19] "The Software-artifact Infrastructure Repository," Available: http://sir.unl.edu/portal/, 15 Feb., 2008