Test Suite Reduction with Selective Redundancy

Dennis Jeffrey
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
jeffreyd@cs.arizona.edu

Neelam Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Abstract

Software testing is a critical part of software development. Test suite sizes may grow significantly with subsequent modifications to the software over time. Due to time and resource constraints for testing, test suite minimization techniques attempt to remove those test cases from the test suite that have become redundant over time since the requirements covered by them are also covered by other test cases in the test suite. Prior work has shown that test suite minimization techniques can severely compromise the fault detection effectiveness of test suites. In this paper, we present a novel approach to test suite reduction that attempts to selectively keep redundant tests in the reduced suites. We implemented our technique by modifying an existing heuristic for test suite minimization. Our experiments show that our approach can significantly improve the fault detection effectiveness of reduced suites without severely affecting the extent of test suite size reduction.

1 Introduction

Software testing and retesting occurs continuously during the software development lifecycle. As software grows and evolves, so too do the accompanying test suites. Over time, some test cases in a test suite may become redundant as the requirements executed by them are also executed by other test cases in the test suite. Due to time and resource constraints for re-testing the software every time it is modified, it is important to develop techniques that keep the test suite size manageable by removing those test cases that may have become redundant with respect to the coverage of program requirements.

Since test suite minimization removes test cases, minimized suites may be weaker at detecting faults in software than their unminimized counterparts. Previous work on test suite minimization has shown some conflicting results. In [18], it was shown that minimizing test suites while keeping all-uses coverage constant could result in little to no loss in fault detection effectiveness. However, the empirical study conducted in [14] suggests that minimized test suites can severely compromise the fault detection capabilities of the test suites. There are two implications of this conflict: first, there are situations where minimization can achieve high suite size reduction without significantly decreasing fault detection effectiveness; second, there are also situations where minimization can achieve high suite size reduction at the expense of significant loss in fault detection effectiveness.

Intuitively, any time a test case is thrown away from a suite, the suite loses an opportunity for detecting faults. Test suite reduction, therefore, ultimately involves a tradeoff between the suite's size and fault detection effectiveness. The focus of the approach for test suite reduction developed in this paper is to achieve high suite size reduction while simultaneously allowing for low fault detection effectiveness loss. The intuition driving our current work is that when a non-reduced suite contains lots of redundancy with respect to a coverage criterion, it may be helpful to selectively keep some of that redundancy in the reduced test suite so as to retain more fault detection effectiveness in the reduced suite, hopefully without significantly affecting the amount of suite size reduction. The *test suite minimization* problem [7] can be stated as follows.

Given: A test suite T, a set of testing requirements $\{r_1, r_2, \cdots, r_n\}$, that must be satisfied to provide the desired test coverage of the program, and subsets $\{T_1, T_2, \cdots, T_n\}$ of T, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i covers r_i.

Problem: Find a *minimal cardinality* subset of T that exercises all r_i's exercised by the unminimized test suite T.

For example, the desired coverage of the program may be a set of test cases that cover all edges in the control flow graph of the program. The test suite minimization problem then is: given a test suite satisfying the all-edges adequacy criterion, find a minimal cardinality subset of the test suite that covers all edges in the program. The existing techniques for test suite minimization do not consider keeping any redundancy with respect to the given coverage criterion during the suite minimization process.

Any test suite minimization algorithm addressing the above problem can be modified to incorporate our approach to generate reduced test suites that selectively retain some of the test cases that are redundant with respect to the given coverage criterion. An algorithm based on a heuristic (referred to as the HGS algorithm from here onwards) to select a representative set of test cases from a test suite, providing the same coverage as the entire test suite, was developed by Harrold, Gupta and Soffa [7]. In this paper, we specifically consider this algorithm for test suite minimization and modify it to implement our approach for test suite reduction with selective redundancy. We present the results of our experiments to evaluate the effectiveness of our approach in generating better quality (in terms of their fault detection capability) reduced suites for the Siemens suite programs [2, 11, 13]. The main contributions of this paper are as follows:

- A novel yet simple approach to test suite reduction with selective redundancy.
- Our experimental results clearly show the potential of our new technique in selecting a small set of redundant test cases which have a high chance of detecting new faults.

The remainder of the paper is organized as follows. In the next section, we motivate our approach with an example. Our algorithm for test suite reduction with selective redundancy is described in section 3. In section 4, we present an experimental study comparing an existing test suite minimization technique with our modified version that takes redundancy into account. In section 5, we discuss the related work. We present the conclusions and our future work in section 6.

2 A Motivational Example

We now present a simple example program to motivate our idea of selectively keeping redundant test cases in a reduced test suite generated by a test suite minimization algorithm. The example program and a corresponding branch coverage adequate test suite T with some redundant test cases with respect to branch coverage are shown in Figure 1.

The branches covered by each test case are marked with an X in the respective columns in Table 1. Since our implementation of our approach to test suite reduction with selective redundancy is based on the HGS test suite minimization algorithm [7], we next briefly present the steps of the HGS algorithm.

1. Initially, all requirements are unmarked.
2. For each requirement that is exercised by only one test case each, add each of these test cases to the minimized suite and mark the requirements covered by the selected test cases.
3. Consider the unmarked requirements in increasing order of the cardinality of the set of test cases exercising a requirement. If several requirements are tied since the sets of test cases exercising them have the same cardinality, select the test case that would mark the highest number of unmarked requirements tied for this cardinality. If multiple such test cases are tied, break the tie in favor of the test case that would mark the highest number of requirements with testing sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case among those tied. Mark the requirements exercised by the selected test. Remove test cases that become redundant as they no longer cover any of the unmarked requirements.
4. Repeat the above step until all testing requirements are marked.

We first illustrate how the HGS algorithm will generate a minimized test suite that covers all the branches of the example program in Figure 1. Branch ${B_1}^T$ is executed only by test case t_1. Also, branch ${B_4}^F$ is executed only by test case t_2. Therefore, test cases t_1 and t_2 are added to the minimized suite and the branches covered by these test cases, ${B_1}^T$, ${B_1}^F$, ${B_2}^T$, ${B_2}^F$, ${B_3}^T$, ${B_3}^F$ and ${B_4}^F$, are marked. This makes test case t_3 redundant since all the branches covered by t_3 are marked. Now only branch ${B_4}^T$ is unmarked. Therefore, either t_4 or t_5 can be selected to cover ${B_4}^T$. Since the tie in this case is broken arbitrarily, let t_4 be selected for adding to the minimized suite. Thus, the minimized branch coverage suite generated by the HGS algorithm for this example is $\{t_1, t_2, t_4\}$. Now, test case t_5 is identified as redundant and the algorithm terminates since all the branches are marked. Note that unselected test case t_3 - identified as redundant for branch coverage - exposes a divide-by-zero error in this example.

Now we illustrate our approach for generating a reduced branch coverage adequate test suite by selectively keeping some redundant test cases in the reduced suite. Let us collect the coverage information for a secondary criterion, such as the all def-use pairs criterion, for all the test cases in the test suite T for the example program in Figure 1. This information is shown in Table 2. Our approach is to *modify the step that identifies redundant test cases* in the test suite minimization algorithm. Specifically, when the HGS algorithm identifies a test case as redundant since all the branches covered by this test case have been marked, we check if this test case is also redundant with respect to the secondary criterion. If this is the case, we identify the test case as redundant. Otherwise, we add the test case to the reduced test suite.

In this example, after t_1 and t_2 are added to the reduced suite by the HGS algorithm, t_3 is identified as redundant with respect to branch coverage since all the branches covered by t_3 are already covered by t_1 and t_2. However, we now check if t_3 is also redundant with respect to the secondary criterion. In this case we find that t_3 covers the def-use pair $x(4, 6)$ that is not covered by either t_1 or t_2. Therefore, we do not identify t_3 as redundant and we add it to the reduced test suite. Thus, in our approach the reduced test suite at this point contains t_1, t_2, and t_3. Now, either one of t_4 or t_5 can be chosen next to cover the branch B_4^T. Let t_4 be added to the reduced test suite. Thus, the requirement B_4^T also gets marked. At this point the test case t_5 becomes redundant with respect to branch coverage as well as with respect to the coverage of the secondary criterion. Therefore, the reduced test suite generated by our approach for this example is $\{t_1, t_2, t_3, t_4\}$. Note that this reduced test suite will expose the divide-by-zero error at line 13.

```
1:    read(a,b,c,d);
B1:   if ( a > 0 )
2:        x = 2;
3:    else
4:        x = 5;
5:    endif
B2:   if ( b > 0)
6:        y = 1 + x;
7:    endif
B3:   if ( c > 0)
B4:       if ( d > 0)
8:            output(x);
9:        else
10:           output(10);
11:       endif
12:   else
13:       output(1/(y-6));
14:   endif
```

A Branch Coverage adequate Suite T

t_1: (a = 1, b = 1, c = -1, d = 0)
t_2: (a = -1, b = -1, c = 1, d = -1)
t_3: (a = -1, b = 1, c = -1, d = 0)
t_4: (a = -1, b = 1, c = 1, d = 1)
t_5: (a = -1, b = -1, c = 1, d = 1)

Figure 1. An example program with a branch coverage adequate test suite T.

Test: Case	B_1^T	B_1^F	B_2^T	B_2^F	B_3^T	B_3^F	B_4^T	B_4^F
t_1:	X		X			X		
t_2:		X		X	X			X
t_3:		X	X			X		
t_4:		X	X		X		X	
t_5:		X		X	X		X	

Table 1. Branch coverage information for test cases in T.

For comparison with our approach explained above, let us apply the HGS algorithm directly to Table 2 to compute a minimized suite with respect to the secondary criterion without retaining redundancy. Test case t_1 will be added to the minimized suite since it covers the def-use pair $x(2, 6)$ that is executed only by t_1. Since t_1 also covers $y(6, 13)$, $a(1, B_1)$, $b(1, B_2)$ and $c(1, B_3)$, these def-use pairs get marked. Now only 3 def-use pairs are unmarked, namely $x(4, 6)$, $x(4, 8)$ and $d(1, B_4)$ which are respectively executed by 2, 2 and 3 test cases. In the next step, we consider the requirements $x(4, 6)$ and $x(4, 8)$ since each of these is executed by test sets of cardinality 2. We now select test t_4 since it executes both $x(4, 6)$ and $x(4, 8)$. At this point all the secondary requirements are marked and the HGS algorithm terminates with the minimized test suite $\{t_1, t_4\}$.

Note that this minimized test suite is not branch coverage adequate since it does not cover the branch B_2^F. Nor does it expose the divide-by-zero error at line 13. This example points out that some branches such as B_2^F that do not define or use a variable may not be covered if the minimized test suite with respect to the secondary criterion is generated. Note that if we applied the HGS algorithm to generate a minimized suite without redundancy for both the branch coverage and the coverage of the secondary criterion simultaneously, then the same minimized suite $\{t_1, t_2, t_4\}$ as generated in minimizing with respect to the branch coverage would be generated. In this case again, the divide-by-zero error at line 13 would be missed.

Therefore, the above example suggests that our approach to test suite reduction with retaining selective redundancy in the reduced test suite may be preferable to either of the non-redundancy minimization approaches. The above example also provides insight into why this is so. The def-use pair $x(4, 6)$ is exercised by both t_3 and t_4. The test case t_3 exercises a combination of branch outcomes not executed by the other test cases and this combination of branch outcomes exposes the divide-by-zero error. However, in both of the above test suite minimization schemes without redundancy, t_3 becomes redundant due to the other test cases that are added to the minimized test suite early on in the minimization algorithm. However, in our approach as soon as a test case becomes redundant according to branch coverage, we add it to the reduced suite if it adds new def-use pair coverage. Therefore, it allows t_3 to be added to the reduced suite before t_4 is added to the test suite.

Thus, while our reduction with redundancy approach achieves slightly less suite size reduction, it is also likely to retain test cases that execute different combinations of branch outcomes than those covered by the test cases already in the minimized suite, even though these test cases have become redundant with respect to branch coverage. Thus, we see that there is something significant that may be gained by approaching the test suite reduction problem with the goal of *adding some*, rather than simply removing all, redundancy.

3 Test Suite Reduction With Selective Redundancy

Our proposed approach to test suite reduction is based on the following key observation. Test suite minimization techniques attempt to throw away test cases that are redun-

test case	x(2,6)	x(4,6)	x(4,8)	y(6,13)	a(1, B_1)	b(1, B_2)	c(1, B_3)	d(1, B_4)
t_1:	X			X	X	X	X	
t_2:					X	X	X	X
t_3:		X		X	X	X	X	
t_4:		X	X		X	X	X	X
t_5:			X		X	X	X	X

Table 2. Definition-use pair coverage information for test cases in T.

dant with respect to the coverage criterion for minimization. However, in the absence of an ideal coverage criterion, throwing away test cases can result in significant loss of fault detection capability [14]. Therefore, we believe that the test suite reduction problem should be viewed from the perspective of keeping redundant test cases that may exercise different situations in program execution even though they are redundant with respect to the coverage criterion for test suite minimization. For making a distinction between the primary coverage criterion used for test suite reduction and additional requirements whose coverage determines if a redundant test case should be added to the reduced suite, we respectively refer to them as *primary* and *secondary* criteria. Our approach is very general and even some requirements derived from black-box testing could be used as secondary requirements in conjunction with the statement or branch coverage criteria that may be used as a primary criterion in this approach.

We developed a specific implementation of our test suite reduction with redundancy algorithm by adding a new step to the HGS heuristic algorithm [7]. Instead of throwing away test cases that are redundant with respect to the primary requirement coverage criterion for test suite minimization in the original HGS algorithm, our new step examines the redundant test cases with respect to the coverage of some additional secondary requirements and uses this information to decide whether to add the test case to the reduced suite. Figures 2, 3 and 4 show our implementation of our approach based on the HGS test suite minimization algorithm.

Set of primary testing requirements: r_1, r_2, ..., r_n.
Set of secondary requirements: r'_1, r'_2, ..., r'_m.
Test cases in unreduced test suite: t_1, t_2, ..., t_{nt}.
Input: T_1, T_2, ..., T_n: test sets for r_1, r_2, ..., r_n respectively.
T'_1, T'_2, ..., T'_m: test sets for r'_1, r'_2, ..., r'_m respectively.
Output: RS: a reduced subset of t_1, t_2, ..., t_{nt}

Figure 2. Input/output for our algorithm.

The input and output of our algorithm are shown above in Figure 2. The main algorithm for test suite reduction with selective redundancy is shown in Figure 3, and Figure 4 shows a function *SelectTests* that is used by the main algorithm. As shown in Figure 2, our algorithm takes as input two collections of associated testing sets. T_1, T_2, ..., T_n are the testing sets corresponding to primary requirements such that T_i contains the set of test cases that cover the primary requirement r_i. Similarly, T'_1, T'_2, ..., T'_m are the testing sets corresponding to secondary requirements such that T'_i contains the set of test cases that cover the secondary requirement r'_i. Now we describe the steps in the main algorithm in Figure 3.

Step 1: Initialization. This step simply initializes the variables and data structures that will be maintained throughout the execution of the algorithm. After initialization, the main program loop begins which attempts to select test cases that cover the primary requirements that are currently uncovered by the reduced suite (initially empty). The uncovered primary requirements are considered in increasing order of associated testing set cardinality.

Step 2: Select the Next Test Case According to the Primary Requirement. The algorithm first collects together all the test cases comprising the testing sets of the current cardinality that are associated with uncovered primary requirements. This is the pool from which the next selected test case (with respect to the primary requirements) will be selected. The algorithm next decides which of the tests in the pool to select by giving preference to the test case that covers the most uncovered requirements whose testing sets are of the current cardinality. In the event of a tie, the algorithm recursively gives preference to the test case among the tied elements that covers the most uncovered requirements whose testing sets are of successively higher cardinalities. If the cardinality reaches the maximum cardinality and there are still ties, an arbitrary test case is selected from among the ties. The selected test case is then added to the reduced suite.

Step 3: Mark Newly-Covered Requirements and Update Coverage Information. At this point, we have added a new test case to the reduced suite. This test case covers certain primary requirements, so the algorithm updates its data structures to reflect the current primary coverage information of the reduced suite. Additionally, if any test case is discovered to become redundant with respect to the primary requirements in this step, then that test case is added to a set of currently-redundant test cases, which will later be examined and from which redundant test cases may possibly be selected for inclusion in the reduced suite. Similarly for the secondary requirements, the algorithm needs to update its data structures to reflect the current secondary coverage information of the reduced suite.

```
Function ReduceWithSelRed($T_1$ ... $T_n$, $T'_1$ ... $T'_m$)
Step1: Unmark all $r_i$ and $r'_i$;
    $redundant$:= {};
    $maxCard$ := maximum cardinality of all $T_i$'s;
    $curCard$ := 0;
    for each test case $t$ do
      $numUnmarked[t]$ := number of $T_i$'s containing $t$;
      $numUnmarked'[t]$ := number of $T'_i$'s containing $t$;
    endfor
Step2: loop
      $curCard$ := $curCard$ + 1;
      while there is a $T_i$ of $curCard$ s.t. $r_i$ is unmarked do
        $list$ := all tests in $T_i$'s of $curCard$ s.t. $r_i$ is unmarked;
        $nextTest$ := SelectTest($curCard$, $list$, $maxCard$);
        $RS$ := $RS \cup \{nextTest\}$;
        $mayReduce$ := FALSE;
Step3:  for each $T_i$ containing $nextTest$ s.t. $r_i$ is unmarked do
          Mark $r_i$.
          for each test case $t$ in $T_i$ do
            $numUnmarked[t]$ := $numUnmarked[t]$ - 1;
            if $numUnmarked[t]$ == 0 AND $t \notin RS$ then
              $redundant$ := $redundant \cup \{t\}$;
          endfor
          if the cardinality of $T_i$ == $maxCard$ then
            $mayReduce$ := TRUE;
        endfor
        for each $T'_i$ containing $nextTest$ s.t. $r'_i$ is unmarked do
          Mark $r'_i$.
          for each test case $t$ in $T'_i$ do
            $numUnmarked'[t]$ := $numUnmarked'[t]$ - 1;
        endfor
Step 4: initialize $count$[t] := 0 for all test cases $t$.
        for each test case $t$ in $redundant$ do
          $count[t]$ := $numUnmarked'[t]$;
        while there is a $t$ in $redundant$ s.t. $count[t] > 0$ do
          $toAdd$ := any $t$ in $redundant$ with max. $count[t]$;
          $RS$ := $RS \cup \{toAdd\}$;
          for each $T'_i$ containing $toAdd$ s.t. $r'_i$ is unmarked do
            Mark $r'_i$.
            for each test case $t$ in $T'_i$ do
              $numUnmarked'[t]$ := $numUnmarked'[t]$ - 1;
          endfor
          initialize $count$[t] := 0 for all test cases $t$.
          $redundant$ := $redundant$ - $\{toAdd\}$;
          for each test case $t$ in $redundant$ do
            $count[t]$ := $numUnmarked'[t]$;
        endwhile
        $redundant$ := {};
        if $mayReduce$ then
          $maxCard$ := maximum cardinality among all $T_i$
              such that $r_i$ is unmarked;
      endwhile
    until $curCard$ = $maxCard$;
end ReduceWithSelRed
```

Figure 3. Algorithm for reduction with selective redundancy.

```
Function SelectTest($size$, $list$, $maxCard$)
    for each test case $t$ in $list$ do
        $count[t]$ := number of unmarked $T_i$'s of
            cardinality $size$ containing $t$;
    $testList$ := test cases $t$ in $list$ s.t. $count[t]$ is maximum.
    if the cardinality of $testList$ == 1 then
        return the test case in $testList$;
    else if $size$ == $maxCard$ then
        return any test case in $testList$;
    else
        return SelectTest($size$+1, $testList$, $maxCard$);
    endif
end SelectTest
```

Figure 4. A function to select the next test case.

Step 4: Select Redundant Test Cases. This step is where redundancy may be added to the reduced suite. For each test case currently known to be redundant with respect to the primary criterion, the number of additional secondary requirements that each test case could add to the coverage of the reduced suite is computed. If some redundant test case adds to the cumulative secondary requirement coverage of the reduced suite, then the test case adding the *most* secondary requirement coverage is selected (ties are broken arbitrarily). The additional secondary requirement coverage of the remaining redundant test cases is recomputed, and this process repeats until either (1) we've selected all the redundant test cases, or (2) no redundant test case adds to the cumulative secondary coverage. At this point, the algorithm has completed processing the current set of redundant test cases. The main algorithm loop iterates again (back to Step 2) until all primary requirements are covered by the reduced suite.

4 Experimental Study

4.1 Subject Programs, Faulty Versions, and Test Case Pools

We followed an experimental set up similar to that used by Rothermel et. al [14]. We used the Siemens programs described in Table 3 as the subject programs. All programs, faulty versions, and test pools used in our experiments were assembled [2, 11, 13] by researchers from Siemens Corporation. We obtained these programs, their faulty versions and the associated test pools from [10]. We examined the types of errors introduced in the faulty versions of each subject program and identified six distinct categories of seeded errors: (1) changing the operator in an expression, (2) changing an operand in an expression, (3) changing the value of a constant, (4) removing code, (5) adding code, and (6) changing the logical behavior of the code (usually involving a few of the other categories of error types simultaneously in one faulty version). Thus, the faulty versions used in our experiments cover a wide variety of fault types. To obtain edge-coverage adequate test suites for each

Name	Lines of Code	Version Count	Description
tcas	138	41	altitude separation
totinfo	346	23	info accumulator
schedule	299	9	priority scheduler
schedule2	297	10	priority scheduler
printtokens	402	7	lexical analyzer
printtokens2	483	10	lexical analyzer
replace	516	32	pattern substitutor

Table 3. Siemens suite subject programs.

program, we randomly selected some number of tests cases from the pool to add to the suite, then added any additional test cases, so long as they increased the cumulative edge coverage, until 100% edge-coverage was obtained. Similar to the experimental set up in [14], the random number 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 constructed 1000 such highly redundant branch coverage adequate test suites for each program. In addition, for each subject program, we also created four more collections of 1000 suites each, where each collection had suite sizes ranging from 0 to 0.4, 0 to 0.3, 0 to 0.2, and 0 to 0.1 times the number of lines of code. Finally, we created one more set of 1000 suites where we simply started with 0 test cases and then added cases as necessary (so long as each increased the cumulative branch coverage) until 100% coverage was obtained. Altogether, therefore, we created 6000 branch coverage adequate test suites for each program comprising 6 different size ranges corresponding to six rows for each program shown in Table 4.

For experiments with our new reduction technique, for each test case we also needed information about the secondary requirements covered by the test case. We chose to use *all-uses* coverage information for each test case computed by the ATAC tool [9] as our secondary criterion. Thus, for each test case, we recorded all the def-use pairs that were covered by the test case (identified as being either computation uses or predicate uses by ATAC). Our motivation for choosing the def-use pair coverage as our secondary criterion is that in general def-use coverage is a stronger criterion than the branch coverage. Therefore, a test case that is redundant with respect to branch coverage may not be redundant with respect to def-use coverage. However, if a weaker criterion such as the statement coverage is selected as the secondary criterion, a test case that is redundant with respect to branch coverage will also be redundant with respect to statement coverage. However, as mentioned before any other criterion that is not subsumed by the primary criterion can also be used as a secondary criterion.

We implemented both the original HGS algorithm and our test suite reduction with selective redundancy (RSR) technique in Java. We conducted the following experiment using the 1000 branch coverage adequate test suites generated for each of the 6 size ranges described above. We *minimized* each of the above branch coverage adequate test suites by applying the original HGS algorithm for removing the test cases redundant with respect to branch coverage. We also *reduced* the above branch coverage adequate suites using our technique by *adding branch coverage redundant test cases* to the reduced suites if they contributed additional def-use pair coverage. The results of our experiments with the above techniques are respectively shown in the columns labeled "HGSBr" (for "branch minimization") and "RSR" (for "reduction with selective redundancy") in Table 4. The table shows the results for each suite size range for each subject program: average original suite size (|T|), average number of faults detected by original suite (|F|), average reduced suite size (|Tmin|), average number of faults detected by reduced suite (|Fmin|), average percentage suite size reduction and average percentage fault detection loss for each of the above techniques. The results for the `printtokens` and `printtokens2` programs are respectively shown in the rows labeled `printtok` and `printtok2`. The values reported in the table are the averages computed across all 1000 suites for each given suite size range for each subject program. For comparison with the above two techniques, we also applied the original HGS algorithm to minimize the above branch coverage adequate suites with respect to their def-use coverage. The results of this experiment are shown in the columns labeled "HGSdu" (for "def-use minimization") in Table 4.

Further, to show that our new RSR technique selects the additional redundant test cases that are effective at detecting new faults, we conducted the following experiment: minimize each suite as done in HGSBr, with one difference: when a minimized suite is computed by the HGSBr technique, we then check whether the corresponding reduced suite computed by RSR is larger or not. If so, we *randomly* add additional tests to the HGSBr-minimized suite until the size matches that of the corresponding RSR-reduced suite. Thus, this experiment computes minimized suites of the same sizes as for technique RSR, but the additional tests selected here are selected randomly, rather than by analysis of the secondary coverage information as is done by RSR. These results are shown for the suites in suite size range 0-0.5 for each program in Table 5. To analyze our results, we present the boxplots[1] in Figures 5 and 6. The

[1]The height of each box in a box plot represents the range of y-values for the middle 50% of the values. The horizontal line within each box represents the median value. The bottom of each box represents the lower quartile, and the top of each box represents the upper quartile. The vertical line stretching below each box ends at the minimum value, and represents the range of the lowest 25% of the values. The vertical line stretching above each box ends at the maximum value, and represents the range of the highest 25% of the values. The average value is depicted by a small x.

Program & Suite Size Range	|T|	|F|	|Tmin|			|Fmin|			% Size Reduction			% Fault Loss		
			HGSBr	RSR	HGSdu	HGSBr	RSR	HGSdu	HGSBr	RSR	HGSdu	HGSBr	RSR	HGSdu
tcas 0	5.71	7.47	5.00	5.16	5.02	6.78	6.92	6.80	11.34	8.87	11.02	8.18	6.39	7.87
tcas 0-0.1	9.56	9.15	5.00	6.20	5.68	6.84	7.46	7.02	41.60	30.18	35.22	22.35	16.53	20.53
tcas 0-0.2	15.20	11.73	5.00	6.94	6.08	6.73	7.83	7.07	57.66	45.54	50.90	37.07	28.56	34.21
tcas 0-0.3	21.39	14.02	5.00	7.32	6.27	6.85	8.25	7.17	66.34	55.23	60.34	44.60	35.62	42.60
tcas 0-0.4	29.07	16.29	5.00	7.71	6.48	6.80	8.56	7.24	73.09	62.95	67.47	52.09	41.79	49.50
tcas 0-0.5	35.63	17.76	5.00	7.91	6.56	6.67	8.59	7.05	76.77	67.57	71.74	56.23	46.13	54.19
totinfo 0	7.30	12.49	5.18	5.47	5.34	11.44	11.87	11.83	26.66	23.06	24.70	7.91	4.77	5.08
totinfo 0-0.1	18.68	14.62	5.11	5.96	5.30	11.44	12.63	12.47	64.58	60.04	63.26	20.31	12.85	13.85
totinfo 0-0.2	35.61	16.73	5.05	6.29	5.19	11.43	13.11	12.84	77.47	73.54	76.71	30.01	20.48	22.03
totinfo 0-0.3	52.07	17.70	5.04	6.44	5.15	11.36	13.19	13.03	82.60	79.21	82.00	34.05	24.05	25.02
totinfo 0-0.4	69.62	18.55	5.04	6.46	5.12	11.42	13.27	13.16	86.48	83.82	86.15	36.92	27.07	27.78
totinfo 0-0.5	87.73	19.16	5.02	6.46	5.09	11.34	13.15	13.16	88.96	86.62	88.68	39.42	30.15	30.21
schedule 0	7.31	3.38	5.11	5.61	5.36	2.88	3.09	2.90	28.70	21.99	25.30	13.57	7.76	13.53
schedule 0-0.1	18.44	4.58	4.99	6.03	5.47	2.89	3.25	2.98	66.77	60.77	63.81	35.05	27.16	33.08
schedule 0-0.2	32.09	5.18	4.98	6.30	5.54	2.81	3.23	2.83	77.29	72.57	75.20	44.63	36.79	44.15
schedule 0-0.3	47.91	5.61	4.86	6.45	5.55	2.91	3.33	2.80	83.29	79.12	81.40	47.39	39.81	49.01
schedule 0-0.4	58.83	5.77	4.78	6.49	5.52	2.87	3.37	2.75	85.03	81.28	83.41	49.35	40.62	51.08
schedule 0-0.5	74.94	5.96	4.74	6.61	5.56	2.88	3.27	2.67	87.91	84.51	86.35	51.18	44.46	54.31
schedule2 0	8.01	2.21	5.37	5.79	4.79	1.89	1.98	1.97	31.51	26.38	39.13	12.43	8.46	8.95
schedule2 0-0.1	18.61	2.57	5.18	6.12	4.83	1.95	2.08	2.04	66.17	60.80	68.78	20.49	15.99	16.93
schedule2 0-0.2	33.19	3.23	5.04	6.23	4.80	1.90	2.13	2.06	77.67	73.53	79.15	36.80	30.37	31.78
schedule2 0-0.3	47.44	3.77	4.94	6.38	4.81	1.89	2.15	2.10	83.29	79.74	84.22	45.07	38.27	39.30
schedule2 0-0.4	61.60	4.35	4.82	6.54	4.88	2.09	2.42	2.25	86.16	82.80	86.66	47.26	40.05	43.60
schedule2 0-0.5	76.34	4.73	4.74	6.71	4.89	2.02	2.44	2.28	88.45	85.36	88.84	51.87	43.15	46.25
printtok 0	15.76	3.38	7.12	7.63	7.44	2.90	3.03	2.98	53.69	50.39	51.61	12.36	9.19	10.32
printtok 0-0.1	27.64	3.64	7.11	7.76	7.49	2.85	3.06	3.04	71.14	68.62	69.62	19.25	14.21	14.68
printtok 0-0.2	46.03	3.96	6.93	7.75	7.38	2.87	3.11	3.05	80.26	78.26	79.12	25.00	19.53	21.04
printtok 0-0.3	63.84	4.28	6.81	7.76	7.29	2.93	3.15	3.09	83.92	82.16	82.95	28.66	24.07	25.12
printtok 0-0.4	83.44	4.54	6.70	7.80	7.26	2.89	3.19	3.11	86.89	85.27	86.01	33.40	27.36	28.97
printtok 0-0.5	101.87	4.75	6.58	7.73	7.17	2.89	3.22	3.15	88.77	87.38	88.03	36.02	29.46	30.88
printtok2 0	11.77	7.36	7.16	9.04	8.78	7.05	7.25	7.24	37.35	21.96	23.96	4.04	1.45	1.51
printtok2 0-0.1	27.56	7.80	6.78	11.79	10.05	7.08	7.49	7.45	68.39	50.02	55.55	8.90	3.82	4.23
printtok2 0-0.2	49.74	8.17	6.25	12.76	10.06	6.99	7.63	7.63	79.76	65.06	70.35	13.94	6.34	6.31
printtok2 0-0.3	75.01	8.45	5.85	13.22	9.92	7.13	7.86	7.78	86.03	73.68	78.56	15.34	6.78	7.66
printtok2 0-0.4	100.34	8.58	5.61	13.41	9.90	7.17	7.89	7.86	88.98	78.57	82.59	16.18	7.82	8.18
printtok2 0-0.5	121.73	8.60	5.49	13.51	9.88	7.13	7.94	7.84	90.19	80.71	84.43	16.72	7.52	8.62
replace 0	18.63	11.13	11.93	14.92	14.53	8.82	10.42	10.33	35.34	19.43	21.50	19.72	6.20	6.92
replace 0-0.1	34.59	14.10	11.75	17.49	15.86	9.03	12.00	11.59	61.18	44.46	48.83	33.98	13.97	16.73
replace 0-0.2	56.67	16.80	11.33	19.13	16.31	8.85	13.12	12.50	73.20	58.45	63.15	44.75	20.49	24.00
replace 0-0.3	82.49	19.01	11.09	20.54	16.70	8.83	13.82	13.06	79.77	66.84	71.45	50.93	25.54	29.25
replace 0-0.4	105.06	19.96	10.90	21.27	16.79	8.77	14.11	13.33	82.35	70.63	74.96	53.04	27.34	31.00
replace 0-0.5	134.59	21.43	10.66	22.39	16.94	8.77	14.53	13.49	86.70	76.10	80.49	56.77	30.38	35.09

Table 4. Experimental results for average percentage suite size reduction and average percentage fault detection loss for different techniques: test suite minimization by the HGS algorithm with respect to branch coverage (HGSBr), test suite reduction with selective redundancy (RSR), and test suite minimization by the HGS algorithm with respect to def-use pair coverage (HGSdu).

boxplots give a statistical view of the data presented in the Table 4. Figure 5 shows the percentage suite size reduction and percentage fault detection loss of the RSR and HGSBr-minimized suites for suite size range 0-0.5. Figure 6 shows the additional-faults-to-additional-test ratio values for those suites in suite size range 0-0.5 where the RSR-reduced suites were larger than the corresponding HGSBr-minimized suites. This ratio is a measure of, for each additional test case in the RSR-reduced suite over the corresponding HGSBr-minimized suite, the number of additional faults detected by the RSR-reduced suite. A value of 1, for instance, would mean that for each additional test in the RSR-reduced suite, 1 more fault is detected. A negative value may occur if the RSR-reduced suite is larger but detects fewer faults than its HGSBr-minimized counterpart. The ratio is computed as the number of additional faults detected by the RSR-reduced suite, divided by the number of additional tests in the RSR-reduced suite.

4.2 Analysis of Results

Reduction in Size of Branch Coverage Adequate Test Suites: We observe from Table 4 that the average sizes of reduced test suites with redundancy generated by RSR were always slightly higher than the average sizes of minimized suites generated with HGSdu, which in turn were slightly higher than the sizes of minimized suites generated with HGSBr. These results are as expected. Neither HGSdu nor HGSBr attempt to retain some test cases that may be redundant with respect to branch coverage or def-use coverage. However, RSR selectively adds those test cases that provide additional def-use coverage at the time they become redundant with respect to branch coverage.

Therefore, RSR reduced suites have their test cases selected in a different order than what is achieved by HGSdu. Therefore, it is expected that the sizes of reduced test suites generated by RSR would be larger than those generated by HGSBr and HGSdu. The white boxes in Figure 5 also show

Program	|Fmin|	% Fault Loss
tcas 0-0.5	8.45	46.55
totinfo 0-0.5	11.96	36.32
schedule 0-0.5	3.26	44.60
schedule2 0-0.5	2.15	49.02
printtokens 0-0.5	2.94	35.12
printtokens2 0-0.5	7.58	11.62
replace 0-0.5	12.13	41.66

Table 5. Avg. number of faults detected and avg. % fault detection loss when randomly selected additional tests are added to match the suite sizes computed by RSR technique.

that while RSR generally achieves less size reduction than HGSBr, both RSR and HGSBr still generally achieve very high levels of suite size reduction.

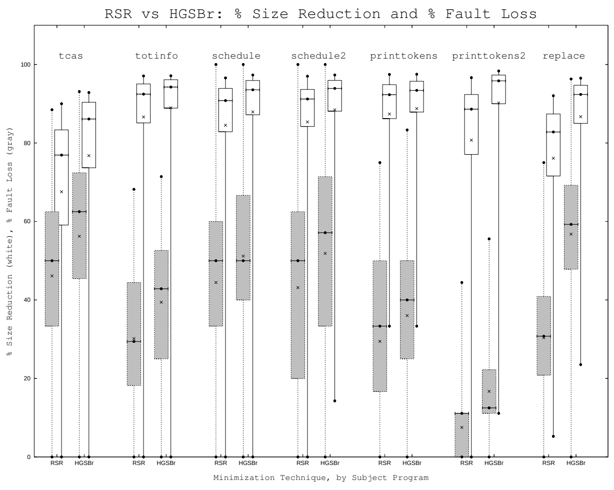

Figure 5. The % suite size reduction (white boxes) and % fault detection loss (gray boxes) for the RSR and HGSBr techniques.

Fault Detection Loss of Reduced Test Suites: We observe from the Table 4 that there is a strong tendency for RSR reduced suites to detect more faults than HGSBr and HGSdu-minimized suites. This is expected since both RSR and HGSdu-reduced suites retain the same all-uses coverage as their non-reduced counterparts, but RSR reduced suites contain some redundancy with respect to test cases that provide additional def-use coverage by exercising already covered branches in a different order. From our experiments it appears that this form of redundancy is effective at retaining test cases that are likely to expose faults. The gray boxes in Figure 5 also show that the fault detection losses experienced by RSR are considerably less than that experienced by HGSBr, and overall, the fault loss values for both RSR and HGSBr are considerably less than the corresponding suite size reduction values.

Test Suite Reduction vs. Fault Detection Loss: HGSBr generally achieves the most suite size reduction at the expense of yielding the most fault detection loss, while RSR

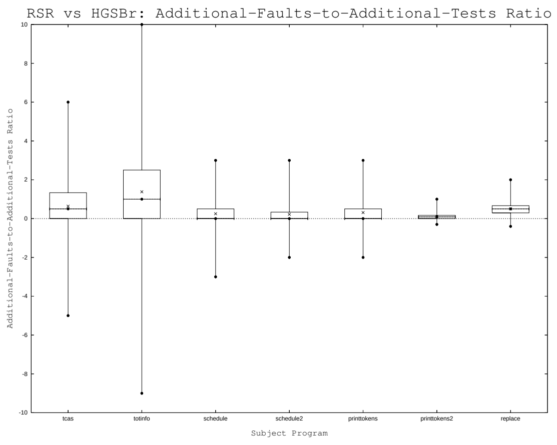

Figure 6. The ratio of additional-faults to additional-tests for the RSR-reduced suites over the HGSBr-minimized suites.

generally achieves the least suite size reduction with the benefit of yielding the least fault detection loss. HGSdu achieves a middle-ground between RSR and HGSBr. Considering that even RSR is still able to achieve relatively high suite size reduction, the benefit of RSR in retaining more fault detection effectiveness in our experiments is evident. Figure 6 shows the benefit of RSR in selecting additional test cases that are likely to expose new faults. From the figure, notice that for every subject program, the average ratio value is above 0. For `tcas`, `totinfo`, `printtokens2`, and `replace`, the median ratio value is above 0, indicating that over half of the ratio values are greater than 0. For `schedule`, `schedule2`, and `printtokens`, although the median value is at 0 with a lower quartile also at 0, indicating that over half of the ratio values are greater than or equal to 0, the average value is more than 0. For `tcas`, the upper quartile is over 1 (more than 25% of suites have ratio value greater than 1), and for `totinfo`, the upper quartile is over 2 (more than 25% of suites have ratio value greater than 2!). For `replace`, even the lower quartile is greater than 0 (over 75% of suites have a positive ratio value).

It is reasonable to ask whether or not the increased fault detection retention of RSR is due *merely* to the fact that the RSR-reduced suites are larger than the other minimized suites. It turns out this is not the case. As indicated by the results in Table 5 compared to the RSR results in Table 4, in all cases, the average number of faults detected by the randomly-added suites is less than the average number of faults detected by the corresponding RSR-reduced suites. Accordingly, the average percentage fault detection loss of the randomly-added suites is always more than the average fault detection loss of the RSR-reduced suites. Our experimental results clearly show the potential of our new technique in selecting a small set of redundant test cases which have a high chance of detecting new faults.

We would now like to elaborate on the following possible points of discussion regarding our approach.

"Fault detection effectiveness loss is still very large across all reduction techniques, even in the new approach - the technique of test suite reduction itself is a lost cause." There are many factors at work which influence the fault detection effectiveness loss of suites. For instance, the test cases used in our experiments were selected from pools assembled by Siemens researchers, and the test cases were generated with respect to various kinds of black-box and white-box approaches. Therefore, many of the test cases in our suites are intentionally meant to test entities within the subject programs that we do not know, nor that we have accounted for during reduction. Therefore, *any* such test cases could be exercising something special about the subject program that we do not realize (such as special boundary conditions or combinations of input values), and thus throwing them away could result in fault detection loss. However, it is quite remarkable that for all the techniques discussed in this paper, much higher percentage suite size reduction is achieved as compared to the corresponding percentage fault detection loss.

"How about the cost of mapping of primary and secondary requirements to test cases?" This consideration is important because the primary motivation for test suite reduction techniques is that testers may be under severe time and resource constraints. However, note that the process of mapping the primary and secondary requirements to test cases is completely automated. On the other hand, the testing process involves more than just executing test cases–the outputs of test cases need to be checked for correctness, which can often be automated only partially or done manually. We believe that the potential savings on time and resource requirements in testing of software, resulting from the use of test suite reduction techniques for periodic maintenance of test suites to keep their size manageable, will offset the cost of mapping the requirements to test cases. It is in this context, our test suite reduction with selective redundancy attempts to retain those test cases in the test suite that are likely to expose faults.

"How does the new approach to reduction with selective redundancy differ from simply using the original HGS algorithm to minimize the test suites with respect to both primary and secondary criteria at the same time?" In our approach, the test cases that become redundant with respect to the primary criterion are checked for their additional coverage with respect to the secondary criterion *as soon as* they become redundant with respect to the primary criterion. So, the reduced suites generated by our approach have their test cases selected in a different order than what is achieved by minimization with respect to only the secondary criterion or both criteria used at the same time. As a result the reduced test suites generated with our approach can have test cases that are redundant not only with respect to the primary criterion but also with respect to the secondary criterion because of the order in which they were added to the reduced suite. The fundamental difference is that our new algorithm specifically seeks to *include redundancy* in the reduced suites while the minimization techniques seek to eliminate as much redundancy as possible.

5 Related Work

Finding a minimal size subset of a test suite that covers the same set of requirements as the unminimized suite is an NP complete problem. This can be easily shown by a polynomial time reduction from the the set-cover problem [6] to the test suite minimization problem. Existing test suite minimization techniques are defined in terms of test case coverage as they attempt to minimize the size of a suite while keeping some coverage requirement constant. A simple greedy algorithm for the set-cover problem (and therefore for the test suite minimization problem) is described in [4]. An algorithm [7] based on a heuristic to select a minimal subset of test cases that covers the same set of requirements as the unminimized suite was developed by Harold, Soffa and Gupta. Agrawal [1] used the notion of megablocks to derive coverage implications among the blocks to reduce test suites such that the coverage of statements and branches in the reduced suite implies the coverage of the rest. Sampath, Mihaylov, Souter and Pollock used concept analysis [16] for incrementally creating and maintaining a reduced test suite for web applications. Although we have implemented our approach to test suite reduction with selective redundancy by modifying the HGS algorithm, our approach is general and can be applied to any test suite minimization technique. A related topic is that of test case prioritization. In contrast to test suite minimization techniques which attempt to remove test cases from the suite, the test case prioritization techniques [5, 15, 17] only re-order the execution of test cases within a suite with the goal of early detection of faults.

In [18], the ATACMIN tool [9] was used to find optimal solutions for minimizations of all test suites examined. This work showed that reducing the size of test suites while keeping all-uses coverage constant could result in little to no loss in fault detection effectiveness. In contrast, the empirical study conducted in [14] suggests that reducing test suites can severely compromise the fault detection capabilities of the suites. The work presented in [8] uses a greedy technique for suite reduction in the context of model-based testing. This work showed that while suite sizes could be greatly reduced, the fault detection capability of the reduced suites was adversely affected. A new model for test suite minimization [3] has been developed that explicitly considers two objectives: minimizing a test suite with respect to a particular level of coverage, while simultaneously trying to maximize error detection rates with respect to one particular fault. A limitation of this model is that fault detection information is considered with respect to a single fault (rather than a collection of faults), and therefore there may be limited confidence that the reduced suite will be useful in detecting a variety of other faults. Techniques for test suite minimization that are specifically tailored to consider the complexity of the modified condition/decision coverage criterion have been developed in [12]. Experimental results in [12] showed that while suite size reduction could be substantial for both reduction techniques, the fault detection loss of suites reduced under these two techniques may vary greatly depending upon the particular program and test suites used.

Suite size and fault detection effectiveness are opposing forces in the sense that more suite size reduction would intuitively imply more fault detection effectiveness loss, since throwing away more test cases, in effect, throws away more opportunities for detecting faults. Thus, there seems to be an inherent tradeoff involved in test suite reduction: one may choose to sacrifice some suite size reduction in order to increase the chances of retaining more fault detection effectiveness. Our experimental results suggest that our new algorithm may provide a framework for testers to have more flexibility in determining the conditions of this tradeoff.

6 Conclusions and Future Work

We have presented a new approach to test suite reduction that attempts to selectively keep redundant test cases with the goal of decreasing the loss of fault detection effectiveness due to reduction in suite size. Our approach is general and can be integrated into any existing test suite minimization algorithm. In our experimental study, our approach consistently performed better than test suite minimization without redundancy by generating reduced test suites with less fault detection loss at the expense of a small increase in the size of the reduced suite. In the near future, we plan to evaluate the effectiveness of our technique for test suite reduction with selective redundancy for a combination of white-box and black-box coverage requirements.

Acknowledgments.
We thank Dr. Gregg Rothermel, Dept. of Computer Science, University of Nebraska, for providing the Siemens suite of programs, their instrumented versions for measuring edge coverage and the associated test suites.

References

[1] H. Agrawal. Efficient coverage testing using global dominator graphs. *Workshop on Program Analysis for Software Tools and Engineering*, pages 11–20, September 1999.

[2] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 210–218, December 1989.

[3] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. *International Conference on Software Engineering*, pages 106–115, May 2004.

[4] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[5] S. Elbaum, A. G. Malishvesky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

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

[7] 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*, 2(3):270–285, July 1993.

[8] M. P. E. Heimdahl and D. George. Test-suite reduction for model-based tests: Effects on test quality and implications for testing. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 176–185, September 2004.

[9] J. R. Horgan and S. A. London. ATAC: A data flow coverage testing tool for C. *Proceedings of Symposium on Assessment of Quality Software Development Tools*, pages 2–10, May 1992.

[10] http://www.cse.unl.edu/$\sim$galileo/sir.

[11] 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*, pages 191–200, May 1994.

[12] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[13] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[14] 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. *International Conference on Software Maintenance*, pages 34–43, November 1998.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions of Software Engineering*, 27(10):929–948, October 2001.

[16] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 132–141, September 2004.

[17] A. Srivastava and J. Thiagrajan. Effectively prioritizing tests in development environment. *International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[18] 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*, 28(4):347–369, April 1998.