Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction

Dennis Jeffrey and Neelam Gupta

Abstract—Software testing is a critical part of software development. As new test cases are generated over time due to software modifications, test suite sizes may grow significantly. Because of time and resource constraints for testing, test suite minimization techniques are needed to remove those test cases from a suite that, due to code modifications over time, have become redundant with respect to the coverage of testing requirements for which they were generated. Prior work has shown that test suite minimization with respect to a given testing criterion can significantly diminish the *fault detection effectiveness* (FDE) of suites. We present a new approach for test suite *reduction* that attempts to use additional coverage information of test cases to selectively keep *some* additional test cases in the reduced suites that are redundant with respect to the testing criteria used for suite minimization, with the goal of improving the FDE retention of the reduced suites. We implemented our approach by modifying an existing heuristic for test suite minimization. Our experiments show that our approach can significantly improve the FDE of reduced test suites without severely affecting the extent of suite size reduction.

Index Terms—Software testing, testing criteria, test suite minimization, test suite reduction, fault detection effectiveness.

1 Introduction

Software testing and retesting occurs continuously during the software development lifecycle. As software grows and evolves, new test cases are generated and added to a test suite to exercise the latest modifications to the software. Over several versions of the development of the software, some test cases in the test suite may become redundant with respect to the testing requirements for which they were generated since these requirements are now also satisfied by other test cases in the suite that were added to cover modifications in the later versions of software. Due to time and resource constraints for retesting the software every time it is modified, it is important to develop techniques that keep test suite sizes manageable by periodically removing redundant test cases. This process is called *test suite minimization*. The test suite minimization problem [11] can be formally stated as follows:

Given. A test suite T of test cases $\{t_1, t_2, t_3, \ldots, t_m\}$, 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_is such that any one of the tests t_j belonging to T_i satisfies r_i.

Problem. Find a *minimal cardinality* subset of T that exercises all r_is exercised by the unminimized test suite T.

In general, the problem of selecting a minimal cardinality subset of T that satisfies all the requirements covered by T is NP-complete since the *minimum set-cover* problem [8] can be reduced to the test suite minimization problem in polynomial time. Therefore, heuristics for solving this problem become important.

A classical greedy heuristic [6], [7] for the minimum set-cover problem is as follows: Pick the test case that covers the most requirements, remove all the requirements covered by the selected test case, and repeat the process until all the requirements are covered. The ties are broken arbitrarily. Another heuristic to minimize test suites, developed by Harrold et al. in [11], greedily selects the next test case exercising the most additional requirements that are satisfied by the fewest number of tests.

The purpose of testing criteria (such as branch coverage or all-uses coverage) is to assess the adequacy of test suites and to provide a check on suite quality. Given a testing criterion C that is satisfied by a test suite T, a test case t in T is *redundant with respect to* C if the smaller suite $T - \{t\}$ also satisfies C. Thus, the process of removing test cases from a test suite that are redundant with respect to certain testing criteria *preserves* the adequacy of the suite *with respect to those criteria*. Some prior empirical studies [22], [23], [31] have used the code coverage criteria for minimizing the test suites. In experiments by Wong et al. [31], minimized test suites achieved 9 percent to 68 percent size reduction while only experiencing 0.19 percent to 6.55 percent fault detection loss. On the other hand, in the empirical study conducted by Rothermel et al. [22], the minimized suites achieved about 80 percent suite size reduction on average while losing about 48 percent fault detection effectiveness (FDE) on average. These results are encouraging as *much higher percentage suite size reduction was achieved as compared to the percentage loss in FDE* of suites.

• *The authors are with the Department of Computer Science, University of Arizona, Tucson, AZ 85721. E-mail: {jeffreyd, ngupta}@cs.arizona.edu.*

Manuscript received 20 Nov. 2005; revised 24 Jan. 2006; accepted 9 Nov. 2006; published online 28 Dec. 2006.
Recommended for acceptance by E. Weyuker.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0310-1105.

```
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)

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	

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

There are a variety of testing criteria that have been discussed in literature, and some are finer (stronger) than others [9]. We observed that different testing criteria are useful for identifying test cases that exercise different structural and functional elements in a program, and we therefore believe the use of multiple testing criteria can be effective at identifying test cases that are likely to expose different faults in software. In this paper, we present a new approach for test suite reduction that makes use of multiple testing criteria (corresponding to multiple types of testing requirements). The key step of our approach is that when a test case t is selected into a reduced suite because it satisfies an additional requirement with respect to some testing criterion C, we then check the following: Among those other test cases R that become redundant with respect to C as a result of the selection of t, we select those test cases from R into the reduced suite that satisfy additional requirements with respect to some *other* testing criterion. Thus, our approach selectively retains those test cases that are redundant with respect to some testing criterion, if those test cases are *not* redundant according to some *other* testing criterion. We call our approach "Reduction with Selective Redundancy (RSR)." We implemented our approach and conducted experiments with several programs to evaluate and compare the effectiveness of our approach with prior experimental studies [22], [23] on test suite minimization. Our experimental results show that our approach can significantly improve the FDE retention of reduced suites over minimized suites at the cost of only a relatively small increase in the sizes of the reduced suites.

The main contributions of this paper are as follows: 1) A novel yet simple approach to test suite reduction that focuses on retaining test cases that may expose different faults in software. 2) Our experimental results clearly show the potential of our new reduction approach, as compared to a minimization approach, in terms of retaining significantly more FDE in reduced suites while still allowing for significant suite size reduction.

The remaining paper is organized as follows: Section 2 motivates our approach with an example. Section 3 contains the outline of our approach, and Section 4 discusses a specific implementation of our approach. Section 5 presents an experimental study that compares the results of our approach with the results obtained when using a minimization approach. Section 6 discusses related work. Finally, the conclusions are mentioned in Section 7.

2 Motivational Example

We now present a simple example program shown in Fig. 1 to motivate our approach. A branch coverage adequate test suite T for the program is also shown. The branches covered by each test case are marked with an X in the respective columns in the table in Fig. 1.

We first show the result of minimizing the test suite in Fig. 1 using the minimization algorithm developed by Harrold et al. [11] (henceforth called the "HGS algorithm"). Given a test suite T and a set of testing requirements $r_1, r_2, \cdots, r_n$ that must be exercised to provide the desired testing coverage of the program, the HGS algorithm considers the subsets $T_1, T_2, \cdots, T_n$ of T such that any one of the test cases t_j belonging to T_i can be used to test r_i. First, all the test cases that occur in T_is of cardinality one are selected in the representative set and the corresponding T_is are marked. Then, T_is of cardinality two are considered. Repeatedly, the test case that occurs in the maximum number of T_is of cardinality two is chosen and added to the representative set. All unmarked T_is containing these test cases are marked. This process is repeated for T_is of cardinality $3, 4, \cdots, max$, where max is the maximum cardinality of the T_is. In case there is a tie among the test cases while considering T_is of cardinality m, the test case that occurs in the maximum number of unmarked T_is of cardinality $m+1$ is chosen. If a decision cannot be made, the T_is with greater cardinality are examined and finally a random choice is made. Consider applying the HGS algorithm to generate a minimized test suite for the example program in Fig. 1. Initially, since branches B_1^T and B_4^F are satisfied only by test cases t_1 and t_2, respectively, both of the tests t_1 and t_2 are selected into the minimized suite. Next, all of the branches satisfied by t_1 and t_2 are marked covered. As a result, test case t_3 becomes redundant with respect to branch coverage since all of its branches are already marked as covered. Now, either t_4 or t_5 can be

TABLE 1
Definition-Use Pair Coverage Information for Test Cases in T

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
All-Uses Coverage Information for Test Cases in T

test case	x(2,6)	x(4,6)	x(4,8)	y(6,13)	a(1,B_1^T)	a(1,B_1^F)	b(1,B_2^T)	b(1,B_2^F)	c(1,B_3^T)	c(1,B_3^F)	d(1,B_4^T)	d(1,B_4^F)
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	

selected to cover the remaining branch B_4^T. Let t_4 be selected into the minimized suite. Branch B_4^T is then marked covered (which causes test t_5 to become redundant according to branch coverage), and the algorithm terminates since all testing requirements are now covered by the test cases in the minimized suite $\{t_1, t_2, t_4\}$. Note that the test case t_3 that exposes a divide-by-zero error at line 13 is not selected into the minimized suite. Thus, the fault detection effectiveness of the suite has been reduced due to test suite minimization.

We next use the above example to illustrate our approach that attempts to *retain* some of the test cases that become redundant according to branch coverage. The definition-use pair coverage[1] information for all the test cases in test suite T for the example program is shown in Table 1. We modify the HGS algorithm by *inserting the following check* after each test case t_i is selected: If any test case t_j becomes redundant according to branch coverage due to the selection of t_i, we also select t_j into the reduced suite *only if* t_j is *not* redundant according to definition-use pair coverage.

In our example, after t_1 and t_2 are selected into the reduced suite by the HGS algorithm, t_3 is identified as redundant with respect to branch coverage. However, t_3 covers the definition-use pair $x(4, 6)$ that is satisfied by neither t_1 nor t_2. Therefore, t_3 is selected. Next, either one of t_4 or t_5 can be selected by the HGS algorithm to cover branch B_4^T. Let t_4 be selected. At this point, test case t_5 becomes redundant with respect to branch coverage as well as definition-use pair coverage, so it is *not* selected, and the algorithm terminates since all branches (and all definition-use pairs) are marked as covered at this point. The computed reduced suite is $\{t_1, t_2, t_3, t_4\}$, which exposes the divide-by-zero error at line 13.

If we applied the original HGS algorithm to Table 1 to compute a minimized suite with respect to only definition-use pair coverage, the algorithm would compute the minimized test suite $\{t_1, t_4\}$. Note that this suite is not branch coverage adequate since it does not cover branches B_2^F or B_4^F, nor does it expose the divide-by-zero error at line 13. Further, if we took the union of the set of branches and the set of definition-use pairs and applied the original HGS algorithm with respect to this single set of (combined) testing requirements, then the minimized suite $\{t_1, t_2, t_4\}$ would be computed, which again would not expose the divide-by-zero error at line 13.

Note that the above definition-use pair coverage criterion differs from Rapps and Weyuker's [21] *all-uses*[2] criterion in how *all-p-uses* are defined. In the *all-p-uses* coverage criterion, a predicate use is associated with an *outcome* of the predicate. The *c-uses* are defined in the same way as in the above definition-use pair criterion. The all-uses exercised by each test case for the example program are shown in Table 2. Note that each predicate use in the table now corresponds to two testing requirements: one each for the $true$ and $false$ branch outcomes. For our example, the HGS algorithm computes the minimized suite $\{t_1, t_2, t_4\}$ when minimizing T with respect to the all-uses requirements in Table 2.

Overall, our example suggests that our approach to test suite reduction with retaining selective redundancy while minimizing suites may be preferable to the approaches that minimize a suite with respect to a testing criterion. In all of the above examples of minimization using the HGS algorithm, t_3 becomes redundant and is therefore never selected, due to the other test cases that are selected into the minimized suite early on. However, our approach allows t_3 to be selected into the reduced suite and expose a fault not exposed by other test cases since it executes a different combination of branch outcomes and definition-use pairs than other test cases, while at the same time allowing some degree of suite size reduction to occur.

3 Reduction with Selective Redundancy

The key idea of our approach is that after *each* test case is selected into the reduced suite according to some testing

1. We define a *definition-use pair* to be the triple "var(def,use)," where "var" is the variable name, "def" is the line number of the variable's definition, and "use" is the line number of the variable's use of the associated defined value. Note that this definition-use pair coverage criterion does *not* subsume the branch coverage criterion, since there may exist branches that do not define or use any variables, such as an "if" statement that does not contain an "else."

2. A test suite T satisfies the *all-uses* criterion if for every definition of a variable x, all of its *p-uses* and *c-uses* are covered.

```
input:
   A test suite T
   Requirement coverage information for each test case in T, for testing criteria C_1, C_2, ..., C_k (k >= 2)
output:
   RS: a reduced set of test cases from T that satisfies all testing requirements for the k criteria

algorithm ReduceWithSelectiveRedundancy
        RS := {};
        for each criterion C_{1..k}, label all associated testing requirements as unmarked;
        while T is not empty do
Step 1:     nextTest := the next test case selected w.r.t. C_1 for inclusion in RS;
            RS := RS ∪ {nextTest};
            T := T – {nextTest};
            for each criterion C_{1..k}, label as marked the testing requirements satisfied by nextTest;
            redundant := the set of test cases from T that have just become redundant w.r.t. C_1.
            T := T – redundant;
Step 2:     SelectRedundantTests(RS, redundant, C_2);
        endwhile
        return RS;
end ReduceWithSelectiveRedundancy



function SelectRedundantTests(RS, redundant, C_i)
   while ∃ test t in redundant s.t. t contributes additional C_i coverage to RS do
      toAdd := the test case in redundant contributing maximum additional C_i coverage to RS;
      RS := RS ∪ {toAdd};
      redundant := redundant – {toAdd};
      for each criterion C_{i..k}, label as marked the testing requirements satisfied by toAdd;
      if i < k then
         redundantAgain := the set of test cases from redundant that have just become redundant w.r.t C_i
         redundant := redundant – redundantAgain;
         SelectRedundantTests(RS, redundantAgain, C_{i+1});
      endif
   endwhile
end SelectRedundantTests
```

Fig. 2. Pseudocode for our general approach to reduction with selective redundancy.

criterion C, we use *other* testing criteria to select additional test cases that are redundant with respect to C but that are *not* redundant with respect to the other criteria. The main steps of our approach are shown in Fig. 2. The input is a set T of test cases along with the set of testing requirements satisfied by each test case for at least two different testing criteria. The output is a reduced test suite RS that satisfies all testing requirements satisfied by the original suite. Initially, the set RS is empty, and every testing requirement for each criterion being considered is labeled as *unmarked*.

Step 1. In this step, the *next test case is selected* into the reduced suite according to the testing criterion C_1. The implementation of this step will vary depending upon the suite minimization algorithm used to implement our approach. The testing requirements satisfied by the selected test case are *marked* and the set of other test cases that become redundant with respect to the first criterion as a result of selecting the above test case are recorded.

Step 2. In this step, the *SelectRedundantTests* function is used to select test cases from those that become redundant with respect to criterion C_{i-1}, $i = 2, \ldots, k$. It uses the coverage information of test cases with respect to the *next* testing criterion C_i to select the test case contributing the most additional coverage with respect to C_i. After the function completes, control returns to Step 1 above and this repeats until all requirements are *marked*.

Notice that our approach is independent of the type of testing criteria being considered. Even requirements generated from black-box testing could be used in conjunction with white-box testing criteria such as branch coverage or definition-use pair coverage. Also, our approach can be implemented on top of any minimization algorithm (e.g., HGS algorithm [11] or the classical greedy approach [6]) that maintains a working list of test cases and incrementally selects test cases one-after-the-other into a reduced suite.

4 Implementation of our Approach

An implementation of our RSR approach (based on the HGS minimization algorithm [11]) is shown in Fig. 3. The input is a test suite T and k sets of *test case sets*, that map each testing requirement for each of k criteria to the set of test cases satisfying that requirement. The output is a reduced set RS of test cases. The steps of our algorithm are as follows.

Step 1: Initialization. All requirements are labeled as *unmarked*. Also, for each test case the algorithm maintains the number of *unmarked* testing requirements satisfied by that test case (for each testing criterion being considered). After initialization, the main loop in the algorithm begins which incrementally selects test cases into the reduced suite one-after-the-other; the loop considers the *unmarked* requirements corresponding to the first criterion (C_1) in increasing order of cardinality of associated test case sets.

Step 2: Select next test using the first criterion. All test cases present in the *unmarked* test case sets of the current cardinality are identified. The function *SelectTest* in Fig. 4 selects a test case that satisfies the most *unmarked* requirements whose test case sets are of the current cardinality and

```
input:
  t_1, t_2, ..., t_nt: test cases in original (unreduced) test suite T
  T_1^c, T_2^c, ..., T_n^c: test case sets covering each of r_1^c, r_2^c, ..., r_n^c respectively, for each criterion 1 ≤ c ≤ k
output:
  RS: a reduced subset of T.

algorithm ReduceWithSelectiveRedundancy(T_1^c, T_2^c, ..., T_n^c: 1 ≤ c ≤ k)
Step 1: for each criterion c, "unmark" all r_i^c;
        redundant := {}; RS := {}; curCard := 0;
        maxCard := max cardinality of all T_i^1's;
        for each test case t and criterion c do
           numUnmarked^c[t] := number of T_i^c's containing t;
        endfor
        loop
Step 2:    curCard := curCard + 1;
           while ∃ T_i^1 of size curCard s.t. r_i^1 is unmarked do
              list := all tests in T_i^1's of size curCard s.t. r_i^1 is unmarked;
              nextTest := SelectTest(curCard, list, maxCard);
              RS := RS ∪ {nextTest}; mayReduce := FALSE;
              for each T_i^1 containing nextTest s.t. r_i^1 is unmarked do
                 "mark" r_i^1;
                 for each test case t in T_i^1 do
                    numUnmarked^1[t] := numUnmarked^1[t] − 1;
                    if numUnmarked^1[t] == 0 and t ∉ RS then
                       redundant := redundant ∪ {t};
                 endfor
                 if cardinality of T_i^1 == maxCard then mayReduce := TRUE;
              endfor
              for each criterion c s.t. 2 ≤ c ≤ k do
                 for each T_i^c containing nextTest s.t. r_i^c is unmarked do
                    "mark" r_i^c;
                    for each test t in T_i^c do numUnmarked^c[t] := numUnmarked^c[t] − 1;
                 endfor
              endfor
Step 3:       SelectRedundantTests(RS, redundant, numUnmarked^c: 1 ≤ c ≤ k, 2);
              redundant := {};
              if mayReduce then maxCard := max cardinality of T_i^1's s.t. r_i^1 is unmarked;
           endwhile
        until curCard == maxCard;
end ReduceWithSelectiveRedundancyHGS
```

Fig. 3. Our implementation for reduction with selective redundancy.

adds it to the reduced set. In the event of a tie, the test case that satisfies the most *unmarked* requirements whose test case sets are of successively higher cardinalities is selected. If the cardinality reaches the maximum cardinality, the tie is broken arbitrarily. For each testing criterion, the *unmarked* requirements satisfied by the selected test case are labeled as *marked*. Also, the test cases that now become redundant with respect to the first testing criterion are added to a set of *redundant* test cases.

Step 3: Select from redundant test cases. From among the test cases redundant with respect to C_1, *SelectRedundantTests* in Fig. 4 is used to select test cases in decreasing order of their additional coverage with respect to the second criterion and add them to the reduced set. The newly satisfied requirements are marked and the algorithm recursively tries to select additional redundant test cases using the remaining testing criteria. After selecting redundant test cases, Steps 2 and 3 are repeated until all testing requirements are marked.

Worst-Case Runtime Analysis. Let k be the number of different testing criteria being considered by our algorithm, and let n denote the maximum number of testing requirements associated with any of the k testing criteria. Let MC denote the maximum cardinality among the test case sets considered across all k testing criteria. The runtime of the original HGS algorithm (for only one testing criterion) is bounded by $O(n^*(n+nt)^*MC)$ [11]. Our algorithm has this complexity plus the additional complexity required to account for the other $k-1$ testing criteria during test suite reduction.

Accounting for the other testing criteria involves three steps: 1) determining the occurrences of test cases in the test case sets, 2) updating coverage information as test cases are selected into the reduced suite, and 3) selecting test cases that are redundant according to one testing criterion but not redundant according to some other criterion. Steps 1 and 2 are done in the same way as is done by the HGS algorithm for a single testing criterion. For Step 3, each test case is considered for redundant selection *at most once* for each testing criterion being considered. Accounting for each of the other $k-1$ criteria, therefore, is of no more complexity than accounting for the first criterion. As a result, the worst-case runtime of our implemented algorithm is bounded by the worst-case runtime of the HGS algorithm times a factor of k because there are k criteria being considered instead of just one: $O(k^*n^*(n+nt)^*MC)$.

5 Experimental Study

We conducted experiments to compare the results of *reducing* suites using our reduction with selective redundancy approach with those of *minimizing* test suites with

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


function SelectRedundantTests(RS, redundant, numUnmarked^i: 1 ≤ i ≤ k, c)
    initialize addCoverage[t] := 0 for all tests t;
    for each test t in redundant do addCoverage[t] := numUnmarked^c[t];
    while ∃ t in redundant s.t. addCoverage[t] > 0 do
        toAdd := any test t in redundant with maximum addCoverage[t];
        RS := RS ∪ {toAdd};
        redundant := redundant − {toAdd};
        redundantAgain := {};
        for each criterion m s.t. c ≤ m ≤ k do
            for each T_i^m containing toAdd s.t. r_i^m is unmarked do
                "mark" r_i^m;
                for each test t in T_i^m do
                    numUnmarked^m[t] := numUnmarked^m[t] − 1;
                    if m == c and numUnmarked^m[t] == 0 and t ∉ RS then
                        redundantAgain := redundantAgain ∪ {t};
                    endif
                endfor
            endfor
        endfor
        redundant := redundant − redundantAgain;
        if c < k then
            SelectRedundantTests(RS, redundantAgain, numUnmarked^i: 1 ≤ i ≤ k, c+1);
        endif
        initialize addCoverage[t] := 0 for all tests t;
        for each test t in redundant do addCoverage[t] := numUnmarked^c[t];
    endwhile
end SelectRedundantTests
```

Fig. 4. Function *SelectTest* to select the next test case according to the first testing criterion and function *SelectRedundantTests* to recursively select tests that are redundant with regard to some criterion.

respect to a testing criterion using the HGS algorithm. The first set of experiments uses the Siemens suite [2], [17] and the Space program [30] in C language. In these experiments, testing requirements for the *white-box testing criteria* such as branch coverage, all-uses coverage, and *subpaths of length 3* were considered for reducing the test suites. Given a trace of exercised branches generated from the execution of a test case, we define *a subpath of length 3* to be the sequence of statements on the path defined by any three consecutive branch outcomes on the path. We consider this criterion since it is a stronger than branch coverage criterion. Note that the choice of using *three* consecutive outcomes was somewhat arbitrary and one can use further stronger criteria such as subpaths of length $k > 3$.

The second set of experiments uses Java programs, each containing a method that operates on a data structure. For these experiments, the testing requirements generated from the *specification* of each of these programs *and* the requirements generated from the code coverage criteria such as branch coverage and all-uses coverage were used to reduce the suites. Thus, in this set of experiments, we tried to use criteria that are very different in the sense that one is used for black-box testing and the other is used for white-box testing of programs.

5.1 Experiments with the Siemens Suite and the Space Program

5.1.1 Experiment Setup

This set of experiments follows a setup similar to that used by Rothermel et al. [22], using the Siemens suite and the Space program (Table 3) along with the test pools and the faulty versions available from [15]. We created branch-coverage adequate test suites for six different suite size ranges referred to as $\mathrm{Br}, \mathrm{Br}+0.1, \cdots, \mathrm{Br}+0.5$ to allow varying levels of redundancy. For creating each suite, we first randomly selected a number $X * LOC$ of test cases from a given test case pool to add to the suite, where LOC is the number of lines of code in the given program and, for each of the above ranges $\mathrm{Br}+0.k$ $(k = 0, 1, \ldots 5)$, X is a random variable in the range $(0 \leq X \leq 0.k)$. Also, we added randomly-selected test cases to each suite *as necessary*, so long as each increased the cumulative branch coverage of the suite, until the test suite became branch coverage adequate. For each of the six suite size ranges, we generated 1,000 test suites and conducted the following three experiments.

Experiment MINbr versus RSR. We used the HGS algorithm [11] to minimize each of the above-mentioned 1,000 suites with respect to branch coverage. We refer to this

TABLE 3
Siemens Suite of Programs and the Space Subject Program

Program Name	Lines of Code	Number of Faulty Versions	Test Case Pool Size	Program Description
tcas	138	41	1608	altitude separation
totinfo	346	23	1052	info accumulator
schedule	299	9	2650	priority scheduler
schedule2	297	10	2710	priority scheduler
printtokens	402	7	4130	lexical analyzer
printtokens2	483	10	4115	lexical analyzer
replace	516	32	5542	pattern substituter
Space	6218	38	13585	array definition language interpreter

technique as the *MINbr* technique. For suite reduction using our (RSR) approach, we used two testing criteria: branch coverage as the first criterion and all-uses coverage as the second criterion. We conducted this experiment to compare the suite size reduction versus fault detection retention of the RSR and MINbr techniques. We measured all-uses coverage using the ATAC tool [13]. We refer to the reduced suites produced by the RSR technique as RSR-reduced suites and the minimized suites produced by the MINbr technique as MINbr-minimized suites.

Experiment ADDRAND. To further analyze the effectiveness of our RSR approach, we reduced suites by the MINbr experiment but then *randomly* added additional test cases as necessary to obtain suites that were the same sizes as the RSR-reduced suites. We refer to these suites as ADDRAND-reduced suites. We compared the fault detection retention of the ADDRAND-reduced suites with the RSR-reduced suites.

Experiment RSR3. In this experiment (called RSR3) we studied the effectiveness of the RSR approach for reducing suites using three criteria: branches as the first, all-uses as the second, and *subpaths of length 3* as the third criterion. Note that the *subpaths of length 3* criterion is *control-flow*-based whereas the all-uses criterion is *data-flow*-based. We refer to these reduced suites as RSR3-reduced suites.

We measured the following from our experiments.

- The *percentage suite size reduction* $= \frac{(|T|-|T_{red}|)}{|T|} * 100$, where $|T|$ is the number of test cases in the original suite and $|T_{red}|$ is the number of test cases in the minimized/reduced suite.
- The *percentage fault detection effectiveness (FDE) loss* $= \frac{(|F|-|F_{red}|)}{|F|} * 100$, where $|F|$ is the number of distinct faults exposed by the original suite, and $|F_{red}|$ is the number of distinct faults exposed by the minimized/reduced suite.
- For the suites in suite size range $\mathrm{Br} + 0.5$ such that the RSR approach computes a larger reduced suite than the corresponding MINbr-minimized suite, the *additional-faults-to-additional-tests ratio*

$$= \frac{(|F_{red}|_{RSR} - |F_{red}|_{MINbr})}{(|T_{red}|_{RSR} - |T_{red}|_{MINbr})}.$$

This ratio is a measure of, for each additional test case selected into an RSR-reduced suite above the number in the corresponding MINbr-minimized suite, the number of additional faults detected by the RSR-reduced suite.

5.1.2 Experiment MINbr versus RSR

The results for this experiment are shown in the columns labeled *MINbr* and *RSR* in Table 4. The values in each row of the table are average values for 1,000 suites in each range. The boxplot[3] in Fig. 5 shows the distribution of the percentage size reduction and percentage fault detection loss of suites in the largest suite size range $(\mathrm{Br} + 0.5)$ for each program.

Suite size reduction. For all programs, *less percentage suite size reduction on average was observed for RSR-reduced suites than the respective MINbr-minimized suites.* This is expected since RSR includes selective branch-coverage redundancy in the reduced suites while MINbr attempts to remove as much branch-coverage redundancy as possible. However, notice also that both approaches still achieve relatively high suite size reduction.

Fault detection loss. For all programs, *less percentage fault detection loss on average was observed for RSR-reduced suites than the respective MINbr-minimized suites.* The results with the RSR technique were usually better for the larger suite size ranges. This is because unreduced suites in these ranges contain significant redundancy with regard to branch coverage and thus present more opportunities to the RSR technique to select test cases that execute different combinations of branch outcomes and all-uses. As seen in Fig. 5, the difference in average percentage fault detection loss between the MINbr and RSR approaches is always about the same or greater than the difference in average percentage suite size reduction. Also, we see that for all programs except sched and ptok2, the median fault detection loss is significantly less for RSR than MINbr. The average percentage fault detection loss for the Space program was considerably less than the Siemens suite. This is likely because the test cases for the Space program were generated randomly to achieve branch coverage adequacy. The test pools for the Siemens suite were created to exercise a variety of black-box and white-box testing requirements. Thus, removing a test case from a test suite for a program in the Siemens suite has a greater chance that fault detection effectiveness will be reduced.

We also used the HGS algorithm to *minimize* suites with respect to the union of branch coverage requirements and all-uses coverage requirements. These results are presented under the columns labeled $\mathrm{B} + \mathrm{U}$ in Table 4. The RSR

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

TABLE 4
Experimental Results for Experiments MINbr and RSR

Program/Suite Size Range	$\|T\|$	$\|F\|$	$\|T_{red}\|$			$\|F_{red}\|$			% Size Reduction			% Fault Loss		
			MINbr	B+U	RSR	MINbr	B+U	RSR	MINbr	B+U	RSR	MINbr	B+U	RSR
tcas Br	5.71	7.47	5.00	5.02	5.16	6.78	6.81	6.92	11.34	11.02	8.87	8.18	7.83	6.39
tcas Br+0.1	9.56	9.15	5.00	5.68	6.20	6.84	6.97	7.46	41.60	35.22	30.18	22.35	20.82	16.53
tcas Br+0.2	15.20	11.73	5.00	6.08	6.94	6.73	7.00	7.83	57.66	50.90	45.54	37.07	34.97	28.56
tcas Br+0.3	21.39	14.02	5.00	6.27	7.32	6.85	7.11	8.25	66.34	60.34	55.23	44.60	42.96	35.62
tcas Br+0.4	29.07	16.29	5.00	6.48	7.71	6.80	7.21	8.56	73.09	67.47	62.95	52.09	49.53	41.79
tcas Br+0.5	35.63	17.76	5.00	6.56	7.91	6.67	7.05	8.59	76.77	71.74	67.57	56.23	54.06	46.13
totinfo Br	7.30	12.49	5.18	5.34	5.47	11.44	11.83	11.87	26.66	24.70	23.06	7.91	5.08	4.77
totinfo Br+0.1	18.68	14.62	5.11	5.30	5.96	11.44	12.43	12.63	64.58	63.26	60.04	20.31	14.13	12.85
totinfo Br+0.2	35.61	16.73	5.05	5.19	6.29	11.43	12.79	13.11	77.47	76.71	73.54	30.01	22.35	20.48
totinfo Br+0.3	52.07	17.70	5.04	5.16	6.44	11.36	13.01	13.19	82.60	81.99	79.21	34.05	25.09	24.05
totinfo Br+0.4	69.62	18.55	5.04	5.12	6.46	11.42	13.20	13.27	86.48	86.15	83.82	36.92	27.51	27.07
totinfo Br+0.5	87.73	19.16	5.02	5.09	6.46	11.34	13.18	13.15	88.96	88.67	86.62	39.42	30.04	30.15
sched Br	7.31	3.38	5.11	5.54	5.61	2.88	3.09	3.09	28.70	22.90	21.99	13.57	8.02	7.76
sched Br+0.1	18.44	4.58	4.99	5.63	6.03	2.89	3.21	3.25	66.77	62.80	60.77	35.05	28.21	27.16
sched Br+0.2	32.09	5.18	4.98	5.74	6.30	2.81	3.16	3.23	77.29	74.39	72.57	44.63	38.22	36.79
sched Br+0.3	47.91	5.61	4.86	5.83	6.45	2.91	3.21	3.33	83.29	80.66	79.12	47.39	42.01	39.81
sched Br+0.4	58.83	5.77	4.78	5.83	6.49	2.87	3.24	3.37	85.03	82.65	81.28	49.35	42.88	40.62
sched Br+0.5	74.94	5.96	4.74	5.88	6.61	2.88	3.19	3.27	87.91	8579	84.51	51.18	45.93	44.46
sched2 Br	8.01	2.21	5.37	5.73	5.79	1.89	1.98	1.98	31.51	27.04	26.38	12.43	8.65	8.46
sched2 Br+0.1	18.61	2.57	5.18	5.77	6.12	1.95	2.05	2.08	66.17	62.62	60.80	20.49	16.99	15.99
sched2 Br+0.2	33.19	3.23	5.04	5.75	6.23	1.90	2.05	2.13	77.67	75.02	73.53	36.80	32.17	30.37
sched2 Br+0.3	47.44	3.77	4.94	5.77	6.38	1.89	2.08	2.15	83.29	81.11	79.74	45.07	39.55	38.27
sched2 Br+0.4	61.60	4.35	4.82	5.84	6.54	2.09	2.28	2.42	86.16	84.04	82.80	47.26	43.14	40.05
sched2 Br+0.5	76.34	4.73	4.74	5.86	6.71	2.02	2.25	2.44	88.45	86.60	85.36	51.87	46.67	43.15
ptok Br	15.76	3.38	7.12	7.51	7.63	2.90	2.99	3.03	53.69	51.15	50.39	12.36	9.90	9.19
ptok Br+0.1	27.64	3.64	7.11	7.56	7.76	2.85	3.05	3.06	71.14	69.34	68.62	19.25	14.50	14.21
ptok Br+0.2	46.03	3.96	6.93	7.44	7.75	2.87	3.06	3.11	80.26	78.95	78.26	25.00	20.62	19.53
ptok Br+0.3	63.84	4.28	6.81	7.36	7.76	2.93	3.09	3.15	83.92	82.77	82.16	28.66	25.16	24.07
ptok Br+0.4	83.44	4.54	6.70	7.32	7.80	2.89	3.12	3.19	86.89	85.89	85.27	33.40	28.65	27.36
ptok Br+0.5	101.87	4.75	6.58	7.23	7.73	2.89	3.15	3.22	88.77	87.91	87.38	36.02	30.73	29.46
ptok2 Br	11.77	7.36	7.16	8.78	9.04	7.05	7.25	7.25	37.35	23.96	21.96	4.04	1.49	1.45
ptok2 Br+0.1	27.56	7.80	6.78	10.05	11.79	7.08	7.45	7.49	68.39	55.54	50.02	8.90	4.24	3.82
ptok2 Br+0.2	49.74	8.17	6.25	10.05	12.76	6.99	7.63	7.63	79.76	70.35	65.06	13.94	6.38	6.34
ptok2 Br+0.3	75.01	8.45	5.85	9.92	13.22	7.13	7.79	7.86	86.03	78.56	73.68	15.34	7.58	6.78
ptok2 Br+0.4	100.34	8.58	5.61	9.90	13.41	7.17	7.84	7.89	88.98	82.59	78.57	16.18	8.40	7.82
ptok2 Br+0.5	121.73	8.60	5.49	9.89	13.51	7.13	7.85	7.94	90.19	84.43	80.71	16.72	8.52	7.52
replace Br	18.63	11.13	11.93	14.53	14.92	8.82	10.32	10.42	35.34	21.50	19.43	19.72	7.11	6.20
replace Br+0.1	34.59	14.10	11.75	15.86	17.49	9.03	11.61	12.00	61.18	48.83	44.46	33.98	16.61	13.97
replace Br+0.2	56.67	16.80	11.33	16.31	19.13	8.85	12.52	13.12	73.20	63.14	58.45	44.75	23.90	20.49
replace Br+0.3	82.49	19.01	11.09	16.70	20.54	8.83	12.98	13.82	79.77	71.45	66.84	50.93	29.6	25.54
replace Br+0.4	105.06	19.96	10.90	16.80	21.27	8.77	13.28	14.11	82.35	74.96	70.63	53.04	31.27	27.34
replace Br+0.5	134.59	21.43	10.66	16.95	22.39	8.77	13.52	14.53	86.70	80.48	76.10	56.77	34.97	30.38
Space Br	154.75	31.12	121.09	126.41	127.12	30.43	30.88	30.89	21.70	18.27	17.81	2.18	0.77	0.74
Space Br+0.1	363.32	31.68	118.90	128.28	132.89	30.42	31.11	31.14	60.45	57.64	56.49	3.93	1.79	1.69
Space Br+0.2	650.35	32.09	116.28	127.25	135.84	30.07	31.07	31.10	74.63	72.58	71.33	6.21	3.15	3.03
Space Br+0.3	959.40	32.48	114.43	126.41	137.58	29.83	31.10	31.20	80.19	78.49	77.31	8.04	4.19	3.89
Space Br+0.4	1243.22	32.77	113.22	125.92	138.46	29.68	31.12	31.24	84.08	82.66	81.57	9.30	4.97	4.61
Space Br+0.5	1559.31	32.93	112.09	125.16	138.84	29.55	31.21	31.26	86.16	84.90	83.89	10.13	5.15	5.00

The average original suite size ($|T|$), the average number of faults exposed by the original suite ($|F|$). The average minimized/reduced suite size ($|T_{red}|$). The average number of faults exposed by the minimized/reduced suite ($|F_{red}|$). The average percentage suite size reduction (% Size Reduction), and the average percentage fault detection loss (% Fault Loss).

technique computes slightly larger reduced suites that are more effective at exposing faults than $\mathrm{B}+\mathrm{U}$. This is because RSR attempts to select branch-coverage-redundant test cases *as soon as they become redundant with respect to branch coverage* during suite reduction; this allows for more test cases to be selected due to their all-uses coverage than when suites are simply minimized by removing as much branch and all-uses coverage redundancy as possible. Moreover, we expect the additional test cases selected by RSR to have a chance of exposing additional faults beyond those exposed by other test cases in the reduced suite. This is because they exercise a different combination of branch outcomes and all-uses (and thus exercise a different program behavior).

To determine whether the improvement in fault detection capability observed for RSR-reduced suites over the MINbr-minimized suites is statistically significant, we conducted a *t test for paired observations*[4] [26]. For each of the 1,000 test suites for suite size range $\mathrm{Br}+0.5$, we created the pair (X, Y), where X is the number of distinct faults exposed by the MINbr-minimized suite and Y is the number of distinct faults exposed by the corresponding RSR-reduced suite. We considered the *null hypothesis* that *there is no difference in the mean number of faults exposed by the RSR-reduced suites and the MINbr-minimized suites*. Table 5 shows the resulting *t values* computed for our *t* test, along with the percentage confidence with which we may reject the null hypothesis. We used as reference a table of critical values presented in [26]. Note that the larger the computed *t value*, the greater confidence we have in rejecting the null hypothesis. For our 999 degrees of freedom, it turns out that for *t values* greater than about 3.3, we can reject the null hypothesis with over 99.9 percent confidence. Thus, the differences in the mean number of faults exposed by the RSR-reduced suites and the MINbr-minimized suites are statistically significant.

Additional-faults-to-additional-tests ratio. Fig. 6 shows the additional-faults-to-additional-tests ratio in boxplot format when comparing the RSR-reduced suites over the corresponding MINbr-minimized suites, for suite size range $\mathrm{Br}+0.5$. For all programs, the *average* ratio value is above 0.

4. Also called a *paired t-test*, this is a statistical method for determining whether there may be any statistically significant difference between the means of two populations, given samples where observations from one sample can be naturally paired with observations from the other sample. The procedure is to formulate a *null hypothesis* that assumes the population means are identical, then compute a *t value* from the paired data samples, which is referenced in a corresponding table of critical values to determine the confidence with which we may reject the null hypothesis.

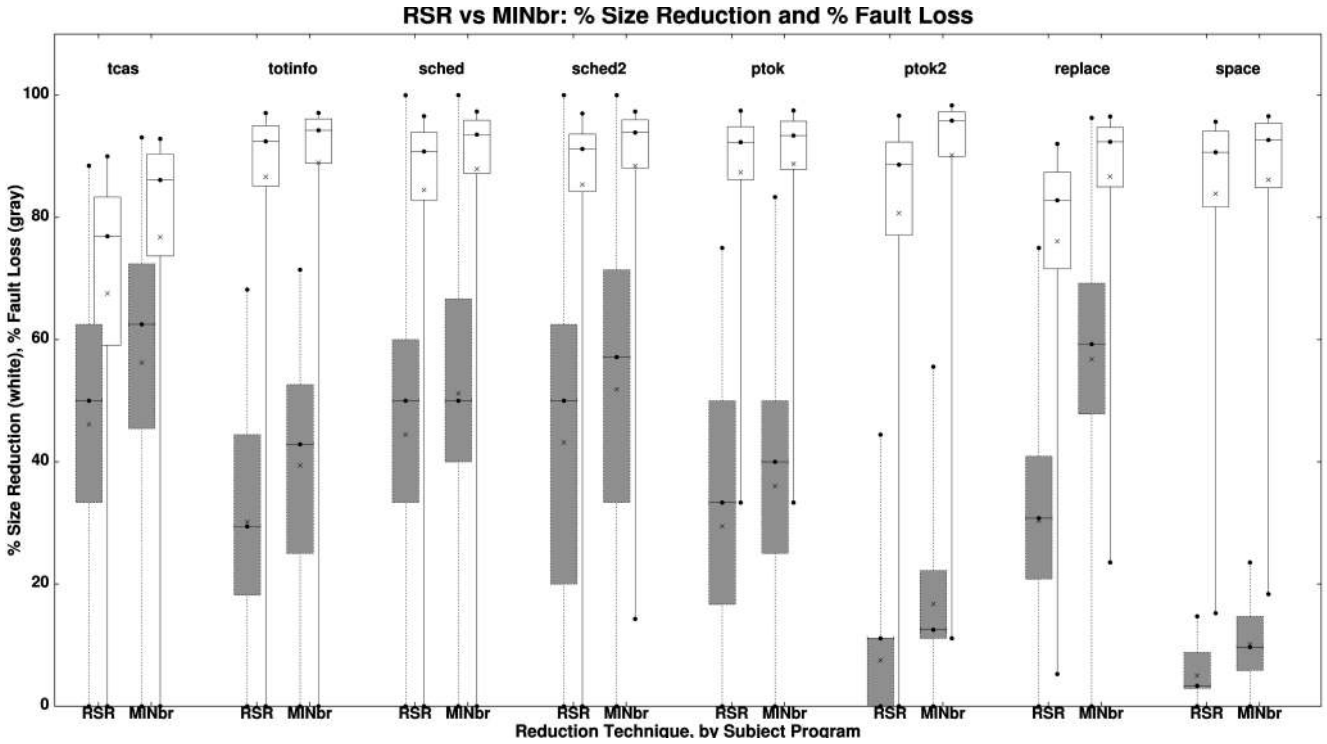

Fig. 5. The percentage suite size reduction (white boxes) and percentage fault detection loss (gray boxes) for RSR (left) and MINbr (right), for suite size range $\mathrm{Br} + 0.5$.

This means that on *average*, each additional test case selected by RSR improved the fault detection capability of the reduced suite. For tcas, totinfo, ptok2, replace, and Space, the *median* ratio value is *above* 0, while for sched, sched2, and ptok, the median value is 0 with the top of the lower quartile also at 0. Note that sched, sched2, and ptok are the three subject programs with the fewest number of faulty versions available (10 or fewer faulty versions each), so for these programs, we can expect many RSR suites to not detect many additional faults simply because there are not many faulty versions available.

For tcas, the bottom of the upper quartile is greater than 1, and for totinfo, the bottom of the upper quartile is greater than 2. For replace, the top of the lower quartile is greater than 0. This suggests for these particular programs RSR was very likely to select the test cases that exposed additional faults. Interestingly, these three particular programs have a relatively higher number of faulty versions available (over 20 each). Even though the Space program has 38 faulty versions available, the median and average ratio value is relatively low at 0.06, and the bottom of the upper quartile occurs at ratio value 0.1. RSR still shows noticeable improvement in fault detection retention on average for the Space program. The results in Fig. 6 suggest that, in general, the additional test cases selected by the RSR approach are likely to improve the fault detection capabilities of reduced suites.

TABLE 5
Computed t Values and the Corresponding Confidence with which the Null Hypothesis Can Be Rejected when Comparing the Number of Faults Exposed by Both the MINbr and RSR Reduced Suites

Program Name	Computed t Value	% Confidence of Rejecting Null Hypothesis
tcas	19.14	>99.9%
totinfo	18.03	>99.9%
sched	12.55	>99.9%
sched2	14.17	>99.9%
ptok	15.22	>99.9%
ptok2	26.66	>99.9%
replace	56.21	>99.9%
Space	33.12	>99.9%

For Suite Size Range $\mathrm{Br} + 0.5$

5.1.3 Experiment ADDRAND

As shown in Table 6, *the average percentage fault detection loss of the ADDRAND-reduced suites was always more than the RSR-reduced suites*. Thus, the RSR approach performed well on average in terms of selecting just those additional tests that are likely to expose additional faults in the software. For programs tcas and sched, the RSR suites were only slightly better on average than the ADDRAND-reduced suites in terms of retaining fault detection. However, for the other five subject programs, the RSR suites achieved between about 3 percent and 11 percent less fault detection loss than the ADDRAND-reduced suites.

Table 7 shows the results of conducting a t test for paired observations comparing the number of distinct faults exposed for both the ADDRAND-reduced suites and the RSR-reduced suites. We can see that, for programs tcas and sched, we do not have strong evidence to reject the null hypothesis. For the other programs, the differences in the average number of faults detected by the RSR-reduced suites and the corresponding ADDRAND-reduced suites were statistically significant.

5.1.4 Experiment RSR3

The results of this experiment are shown in Table 8. In all cases except for some of the smallest suite size ranges, the

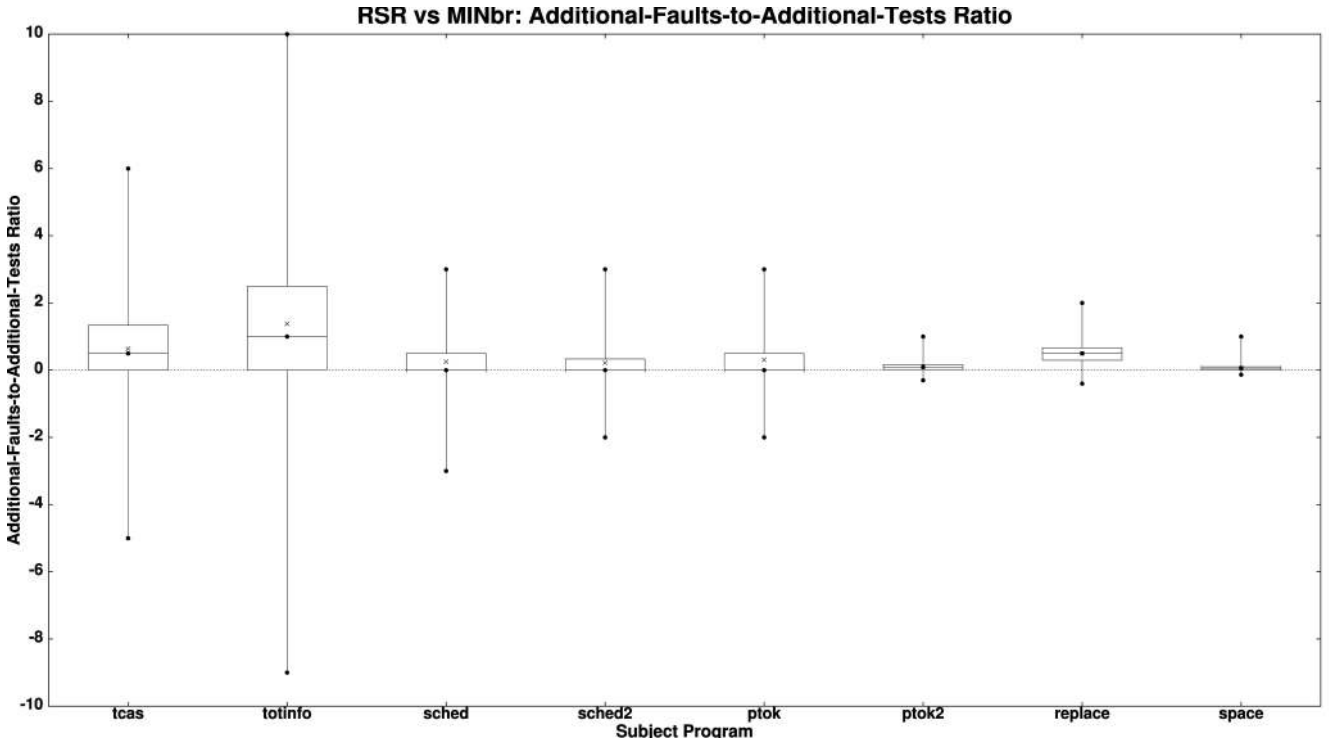

Fig. 6. The additional-faults-to-additional-tests ratio computed from the RSR-reduced suites over the MINbr-minimized suites.

RSR3 approach resulted in less average percentage fault detection loss among suites than the RSR approach. Further, the suite size reduction obtained by RSR3 approach was significant (over 66 percent average reduction in suite size for range $\mathrm{Br} + 0.5$). For program sched, suite size range $\mathrm{Br} + 0.5$, RSR3 resulted in about 10 percent less suite size reduction and over 20 percent less fault detection loss in suites than RSR.

Table 9 shows the results of a t test for paired observations comparing the number of distinct faults exposed for both the RSR-reduced suites and the RSR3-reduced suites. From this table, we see that the null hypothesis can be rejected with high confidence (greater than 95 percent) for all programs except tcas and ptok2. Thus, for most programs, the improvement in the average number of faults exposed by the RSR3-reduced suites when compared with the respective RSR-reduced suites is statistically significant. Note that these results are important considering that some programs such as sched and sched2 have relatively few total faulty versions available, which would limit the amount of improvement by using RSR3 in our experiments. Also, this provides evidence that the benefits of our two-criteria RSR experiment over MINbr are *not* due *merely* to the fact that data flow was used as our secondary criterion. Instead, this suggests that results may be likely to improve when using a second or third criterion *regardless* of whether the additional criteria are data-flow-based or not.

5.2 Experiments with Using RSR to Reduce Test Suites Generated from *Specifications* of Java Data Structure Programs

5.2.1 Experiment Setup

In these experiments we used four programs from [16] mentioned in Table 10, where each program involves a single Java method operating on a data structure. Each Java method

TABLE 6
Experimental Results for Additional Tests Selected Randomly, Compared Against the Corresponding Results for Experiment RSR, Showing the Average Number of Faults Detected by the Reduced Suites ($|F_{red}|$) and the Average Percentage Fault Detection Loss Due to Reduction (Percent Fault Loss)

Program	$\lvert F_{red}\rvert$		% Fault Loss	
	ADDRAND	RSR	ADDRAND	RSR
tcas Br+0.5	8.45	8.59	46.55	46.13
totinfo Br+0.5	11.96	13.15	36.32	30.15
sched Br+0.5	3.26	3.27	44.60	44.46
sched2 Br+0.5	2.15	2.44	49.02	43.15
ptok Br+0.5	2.94	3.22	35.12	29.46
ptok2 Br+0.5	7.58	7.94	11.62	7.52
replace Br+0.5	12.13	14.53	41.66	30.38
Space Br+0.5	30.15	31.26	8.33	5.00

For suite size range $\mathrm{Br} + 0.5$.

TABLE 7
Computed t Values and the Corresponding Confidence with which the Null Hypothesis Can Be Rejected when Comparing the Number of Faults Exposed by Both the ADDRAND and RSR Reduced Suites

Program Name	Computed t Value	% Confidence of Rejecting Null Hypothesis
tcas	1.30	>80.0%
totinfo	11.68	>99.9%
sched	0.26	<50.0%
sched2	8.73	>99.9%
ptok	12.59	>99.9%
ptok2	12.55	>99.9%
replace	23.03	>99.9%
Space	23.78	>99.9%

For Suite Size Range $\mathrm{Br} + 0.5$.

TABLE 8
Results for Experiment RSR3

Program/Suite Size Range	$\|T\|$	$\|F\|$	$\|T_{red}\|$ RSR3	$\|F_{red}\|$ RSR3	% Size Reduction RSR3	% Fault Loss RSR3
tcas Br	5.71	7.47	5.18	6.94	8.52	6.30
tcas Br+0.1	9.56	9.15	6.29	7.45	29.43	16.59
tcas Br+0.2	15.20	11.73	7.08	7.97	44.72	27.66
tcas Br+0.3	21.39	14.02	7.55	8.52	54.32	33.92
tcas Br+0.4	29.07	16.29	7.89	8.66	62.34	41.45
tcas Br+0.5	35.63	17.76	8.14	8.72	66.73	45.27
totinfo Br	7.30	12.49	5.48	11.87	22.92	4.78
totinfo Br+0.1	18.68	14.62	6.11	12.73	59.38	12.24
totinfo Br+0.2	35.61	16.73	6.61	13.27	72.82	19.59
totinfo Br+0.3	52.07	17.70	6.94	13.37	78.35	23.15
totinfo Br+0.4	69.62	18.55	7.19	13.67	82.78	25.09
totinfo Br+0.5	87.73	19.16	7.35	13.60	85.61	27.81
sched Br	7.31	3.38	5.74	3.15	20.39	6.52
sched Br+0.1	18.44	4.58	8.25	3.81	50.41	16.18
sched Br+0.2	32.09	5.18	10.36	4.11	61.00	20.44
sched Br+0.3	47.91	5.61	12.16	4.36	67.68	22.15
sched Br+0.4	58.83	5.77	12.85	4.48	70.24	21.99
sched Br+0.5	74.94	5.96	13.84	4.53	74.25	23.87
sched2 Br	8.01	2.21	5.87	1.98	25.40	8.65
sched2 Br+0.1	18.61	2.57	7.01	2.13	56.57	13.95
sched2 Br+0.2	33.19	3.23	7.67	2.20	69.44	28.47
sched2 Br+0.3	47.44	3.77	8.62	2.36	75.14	33.56
sched2 Br+0.4	61.60	4.35	9.32	2.65	78.18	35.43
sched2 Br+0.5	76.34	4.73	10.10	2.75	80.93	37.82
ptok Br	15.76	3.38	7.65	3.02	50.27	9.43
ptok Br+0.1	27.64	3.64	7.78	3.08	68.56	13.85
ptok Br+0.2	46.03	3.96	7.82	3.12	78.14	19.47
ptok Br+0.3	63.84	4.28	7.90	3.18	81.94	23.45
ptok Br+0.4	83.44	4.54	7.98	3.24	85.09	26.26
ptok Br+0.5	101.87	4.75	8.01	3.26	87.14	28.72
ptok2 Br	11.77	7.36	9.06	7.25	21.84	1.40
ptok2 Br+0.1	27.56	7.80	11.87	7.50	49.80	3.57
ptok2 Br+0.2	49.74	8.17	12.98	7.65	64.70	6.06
ptok2 Br+0.3	75.01	8.45	13.48	7.86	73.30	6.75
ptok2 Br+0.4	100.34	8.58	13.73	7.94	78.18	7.23
ptok2 Br+0.5	121.73	8.60	13.84	7.95	80.44	7.32
replace Br	18.63	11.13	15.25	10.55	17.76	5.21
replace Br+0.1	34.59	14.10	19.07	12.45	40.42	11.21
replace Br+0.2	56.67	16.80	22.22	14.03	53.41	15.48
replace Br+0.3	82.49	19.01	24.84	15.16	61.91	19.07
replace Br+0.4	105.06	19.96	26.44	15.70	65.85	19.97
replace Br+0.5	134.59	21.43	28.82	16.58	71.22	21.45
Space Br	154.75	31.12	127.53	30.89	17.54	0.74
Space Br+0.1	363.32	31.68	136.61	31.17	55.57	1.58
Space Br+0.2	650.35	32.09	142.74	31.19	70.29	2.76
Space Br+0.3	959.40	32.48	146.93	31.31	76.32	3.55
Space Br+0.4	1243.22	32.77	149.80	31.38	80.62	4.17
Space Br+0.5	1559.31	32.93	152.16	31.50	82.97	4.29

The data in this table is organized similarly to Table 4.

is associated with a *precondition* stating that the input data structure must be valid. This precondition is checked by a "repOk" function [4] written in Java that returns *true* or *false* depending upon whether the state of the input data structure is valid. For each Java method, the Korat tool [4] was used to generate all nonisomorphic valid input data structures up to a bounded small size. Since the "specification" of valid input data structure for each Java method is provided by its "repOk" function, we considered the coverage of branches and the definition-use pairs of the "repOk" function as the black-box testing requirements for the respective Java method. These testing requirements were considered as the primary set of requirements with respect to which the test suites were minimized in this experiment. From the pool of test cases generated for each Java method using the Korat tool, we selected test cases to create test suites that were adequate with respect to the above black-box testing requirements. Overall, we generated 1,000 suites for each of six different suite size ranges for each program in the same manner as was done for the Siemens programs and Space. In order to apply our RSR technique, we needed coverage information for each test case for an additional (secondary) testing criterion. For this, we simply recorded the branches and the definition-use pairs covered in each "Java method" (not the "repOk" function, which specified valid input for the Java method) exercised by each test case. Note that in this experiment, we used the RSR technique with two different types of testing requirements resulting from *very different* testing criteria (specification coverage and code coverage of each Java method).

TABLE 9
Computed t Values and the Corresponding Confidence with which the Null Hypothesis Can Be Rejected when Comparing the Number of Faults Exposed by Both the RSR and RSR3 Reduced Suites

Program Name	Computed t Value	% Confidence of Rejecting Null Hypothesis
tcas	1.38	>80.0%
totinfo	5.77	>99.9%
sched	30.09	>99.9%
sched2	10.75	>99.9%
ptok	2.21	>95.0%
ptok2	1.08	>60.0%
replace	27.11	>99.9%
Space	9.47	>99.9%

For suite size range $\mathrm{Br} + 0.5$.

We used the *Java Coverage Analyzer* tool obtained from [14] to record (for each test case for each Java subject) both the branch coverage of the Java subject program as well as the branch coverage of the corresponding "repOk" function. To measure definition-use pair coverage, we hand-instrumented each program to record this information. For each experimental subject, we created faulty versions by seeding errors that were similar in type to those introduced into the Siemens programs (operator change, operand change, constant value change, missing code, added code, logic change). Each faulty version contained a single seeded error, and we attempted to introduce errors that, as in the Siemens faulty versions, would only *sometimes* lead to an exposed error when traversed by a test case.

Experiment minB versus rsrB/W. In the minB experiment, we used the HGS algorithm to minimize the test suites with respect to only the black-box (B) testing requirements generated from coverage of the "repOk" function. For suite reduction using the RSR approach, the set of black-box testing requirements used in minB as described above was considered as the primary criterion, and the set of white-box testing requirements (branches and def-use pairs in the Java method code) covered by each test case were considered as the secondary requirements. We refer to this experiment as rsrB/W.

5.2.2 Experimental Results, Analysis, and Discussion

Suite size reduction and fault detection loss. Table 11 shows the average size reduction and fault detection loss for the reduced suites for experiments minB and rsrB/W. Also, Fig. 7 shows the distribution of percentage size reduction and percentage fault detection loss among suites in the largest suite size ranges for rsrB/W versus minB. For all programs, the rsrB/W experiment resulted in less size reduction but also less percentage fault detection loss on average than the minB experiment. This is because minB does not take into consideration the white-box requirement coverage, while the rsrB/W approach takes into account

TABLE 10
Java Data Structure Experimental Subjects

Program Name	Lines of Code	Number of Faulty Versions	Test Case Pool Size	Program Description
bst	52	13	17647	*remove* from a binary search tree
avl	86	22	2653	*insert* into an AVL tree
heap	36	9	2612	*delete min* from a binary heap
sort	55	14	137257	*quicksort* an array

TABLE 11
Experimental Results for Experiments minB and rsrB/W

Program & Size Range	$\lvert T\rvert$	$\lvert F\rvert$	$\lvert T_{red}\rvert$		$\lvert F_{red}\rvert$		% Size Reduction		% Fault Loss	
			minB	rsrB/W	minB	rsrB/W	minB	rsrB/W	minB	rsrB/W
bst B	5.45	7.67	4.51	4.64	6.94	7.21	15.79	13.60	8.68	5.54
bst B+.1	6.13	8.07	4.47	4.69	6.81	7.31	25.08	21.79	14.44	8.92
bst B+.2	7.88	8.79	4.36	4.72	6.76	7.63	39.35	35.10	21.47	12.59
bst B+.3	10.10	9.40	4.27	4.79	6.77	7.96	49.86	45.12	25.94	14.54
bst B+.4	12.72	9.91	4.21	4.85	6.67	8.18	57.57	52.87	30.42	16.75
bst B+.5	15.35	10.19	4.19	4.89	6.63	8.30	61.89	57.37	32.62	17.80
avl B	6.51	13.87	5.17	5.97	12.74	13.85	19.59	7.73	7.28	0.18
avl B+.1	8.14	14.79	5.19	6.69	12.88	14.75	33.16	15.98	11.70	0.33
avl B+.2	11.99	16.29	5.13	7.83	12.76	16.23	50.18	28.95	20.13	0.39
avl B+.3	15.32	16.77	5.10	8.35	12.67	16.71	58.32	37.55	22.76	0.40
avl B+.4	19.63	17.70	5.07	8.78	12.74	17.66	65.36	45.83	26.54	0.24
avl B+.5	23.95	18.09	5.08	9.15	12.63	18.04	69.97	51.87	28.61	0.28
heap B	1.01	5.15	1.00	1.00	5.15	5.15	0.30	0.05	0.06	0.00
heap B+.1	1.74	6.25	1.00	1.47	5.24	6.15	29.22	10.15	14.22	1.67
heap B+.2	3.70	7.16	1.00	2.17	5.22	6.94	55.88	29.57	24.62	2.91
heap B+.3	5.05	7.60	1.00	2.43	5.25	7.35	64.58	37.97	28.73	3.15
heap B+.4	7.21	7.93	1.00	2.67	5.12	7.57	72.35	48.37	33.52	4.21
heap B+.5	8.66	8.07	1.00	2.74	5.41	7.66	75.09	53.07	31.26	4.75
sort B	1.00	9.61	1.00	1.00	9.61	9.61	0.00	0.00	0.00	0.00
sort B+.1	2.64	11.74	1.00	2.24	9.59	11.68	45.11	9.25	16.53	0.53
sort B+.2	5.57	12.79	1.00	3.14	9.59	12.71	66.39	30.22	23.62	0.57
sort B+.3	7.91	13.10	1.00	3.41	9.59	13.01	73.46	41.20	25.55	0.67
sort B+.4	11.36	13.35	1.00	3.64	9.59	13.25	80.05	52.74	27.15	0.69
sort B+.5	13.51	13.43	1.00	3.63	9.53	13.32	81.81	57.06	28.20	0.83

The format of the data table is similar to that used for describing the results of the Siemens programs and Space.

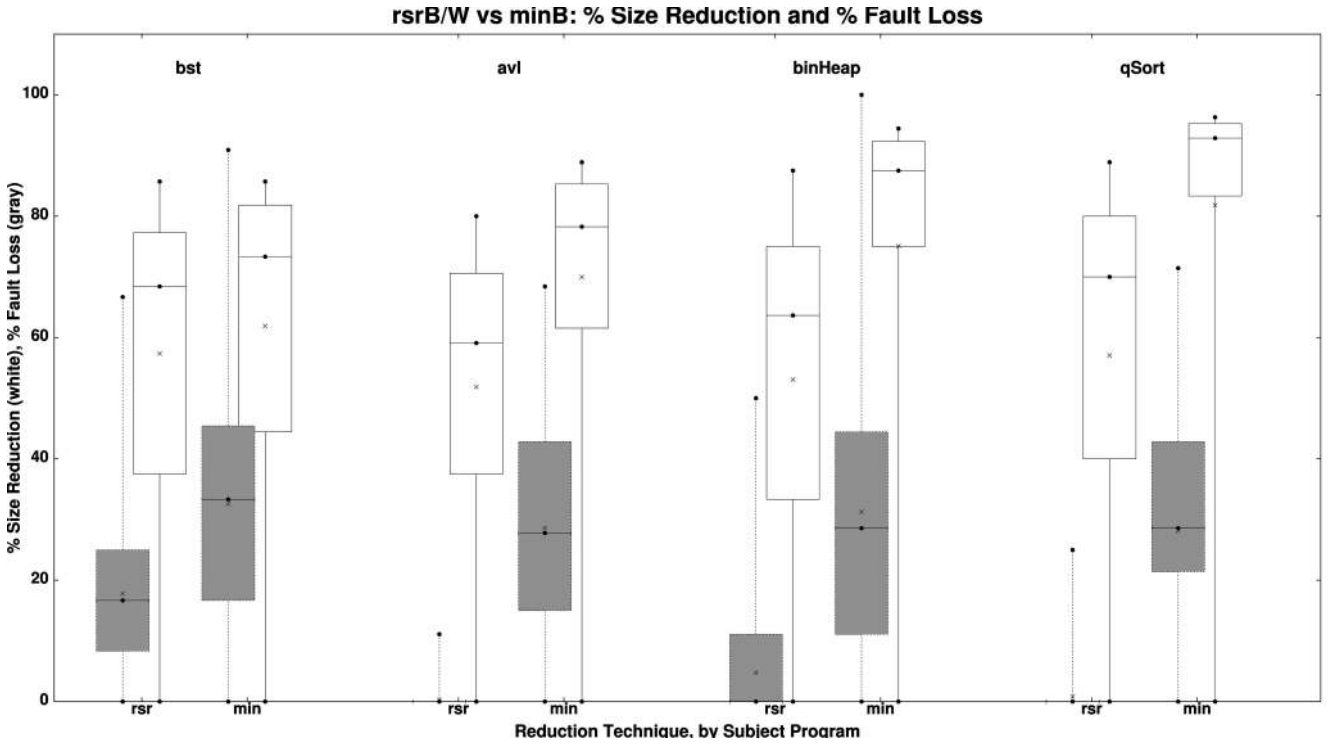

Fig. 7. The percentage suite size reduction (white boxes) and percentage fault detection loss (gray boxes) for rsrB/W (left) and minB (right).

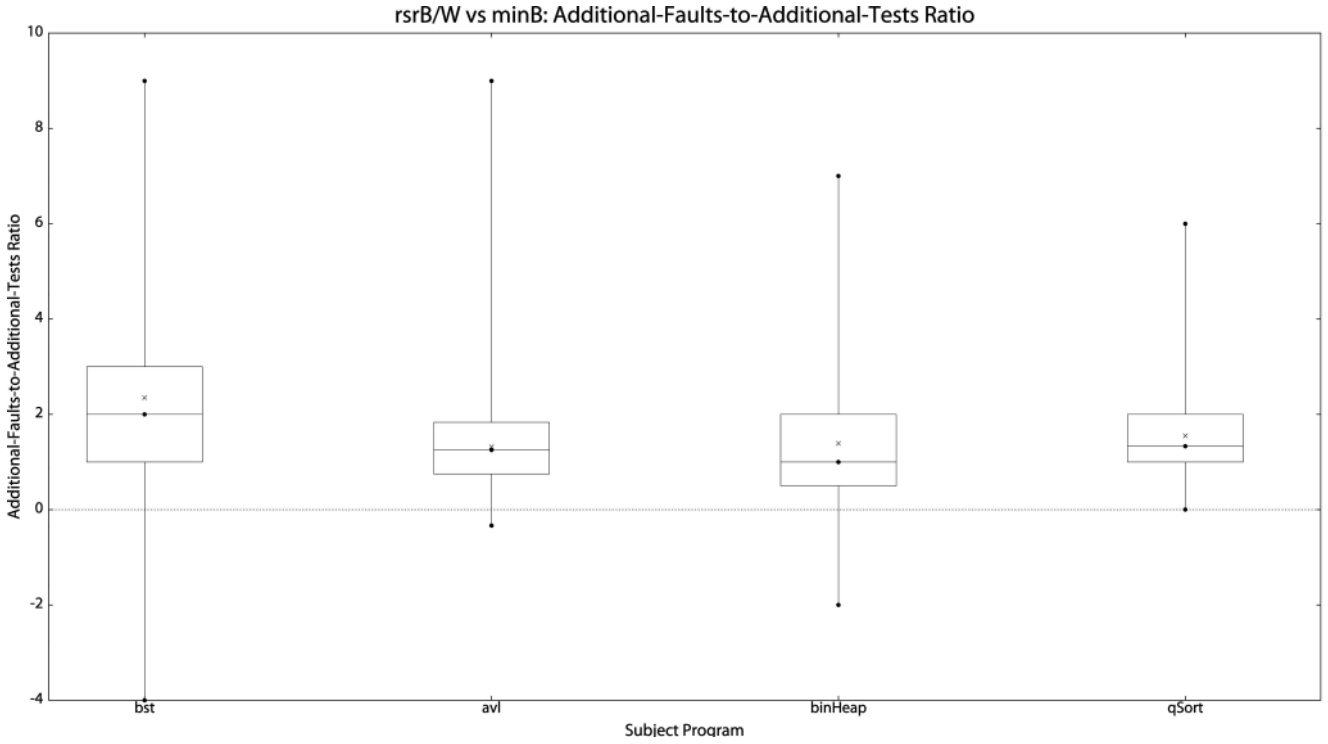

Fig. 8. The additional-faults-to-additional-tests ratio when comparing rsrB/W versus minB.

both the black-box and white-box testing requirements. In fact, on further analysis, we found that many suites minimized with respect to only black-box testing requirements covered fewer branches in the actual code than their unminimized counterparts, and this led to decreased fault detection capability of minB-reduced suites.

Additional-faults-to-additional-tests ratio. Fig. 8 shows the additional-faults-to-additional-tests ratio for the largest suite size range for each program, for those test suites in which rsrB/W computed a larger reduced suite than that computed by minB. From this figure, we notice that in terms of quartiles, for bst, the bottom of the upper quartile is at ratio value 3.00. Also, for all four subject programs, the top of the lower quartile occurs at least at ratio value 0.5. Also note that, for all four subject programs, the *average* ratio value is at least 1.32 and the *median* ratio value is at

least 1.00. Therefore, each additional test case that was selected by rsrB/W in the reduced suite (when compared with the size of the respective minB-minimized suite) exposed more than one additional fault on average. These experiments give us an insight that the *test cases that exercise different combinations of testing requirements corresponding to multiple types of testing criteria are likely to expose different faults*.

5.3 Experimental Conclusions and Threats to Validity

Our experimental results show that using multiple testing criteria during suite reduction is indeed useful in identifying test cases that are likely to expose different faults, as evidenced by the improved fault detection retention of suites using RSR versus MINbr, RSR3 versus RSR, and rsrB/W versus minB. However, we would like to mention some factors that may influence the validity of our results. In our experiments, we do not control for the structure of the subject programs or for the locations where errors are seeded in the faulty versions. Further, the errors in the faulty versions may or may not be representative of errors that typically occur in practice. Also, the set of programs used in our experiments may or may not be representative of other programs that are used in practice. To account for these, we conducted experiments on a variety of programs (written in both C and Java) with a variety of associated faulty versions, and we also included results for the relatively large Space program.

5.4 Additional Points of Discussion

The fault detection loss for the Siemens suite programs was found to be still relatively large across all reduction techniques. The test cases for the Siemens programs were generated with respect to various kinds of black-box and white-box approaches. Therefore, many of these tests are intentionally meant to cover entities in the programs that we do not know, nor that we have accounted for during reduction. Thus, throwing them away could result in fault detection loss. However, as our experiments show, using multiple criteria in test suite reduction can significantly increase the fault detection effectiveness of the reduced suite.

Another issue is the cost of mapping the primary and secondary requirements to test cases. Computing this mapping can usually be automated. Further, the testing process involves more than just executing test cases; outputs of test cases need to be checked for correctness, which can often be only partially automated or must be done manually. We believe that the potential savings of time and resources in the testing of software, resulting from the use of test suite reduction techniques, will offset the cost of mapping the requirements to test cases. In addition, if the developer is not interested in throwing away test cases, the test cases in the reduced suites could be scheduled to be executed ahead (refer to the topic of *test case prioritization* in the Related Work section) of the other test cases in the suite in order to expose a large number of faults early on in the testing.

6 Related Work

Related work can be classified into that which proposes new minimization techniques and that which focuses on conducting empirical studies using existing minimization techniques.

Minimization Techniques. The classical *greedy heuristic* for solving the set-cover problem was presented by Chvatal [6]. The approach greedily selects the next set (test case) that maximizes the ratio of additional requirement coverage to cost, until no sets provide any additional requirement coverage. Another heuristic presented by Harrold et al. [11] (the *HGS algorithm*) greedily selects the next test case exercising the most additional requirements that are satisfied by the fewest number of tests. Chen and Lau [5] described two strategies for dividing a test suite into k smaller subproblems (subsuites) such that *if* optimal solutions can be found for each of the k subproblems, *then* these solutions can be combined to form an optimally reduced suite. However, these two dividing strategies cannot be applied to every suite. Agrawal [1] developed a technique using global dominator graphs to derive implications among testing requirements such that satisfying one requirement implies satisfying one or more of the other requirements. These implications can be used to achieve higher coverage with smaller suites by targeting those requirements implying the most coverage of the other requirements. Tallam and Gupta [28] developed another heuristic called *Delayed-greedy* that exploits *both* the implications among test cases and the implications among the requirements to remove the implied rows and columns in the table mapping test cases to the requirements covered by them. It delays the application of the greedy heuristic until after the table cannot be reduced any further and after the essential tests are selected. Selecting a test case using the greedy heuristic and removing the corresponding row and the columns from the table exposes new implications among test cases and the implications among the requirements, which enables further reduction of the table. All the above heuristics to generate a minimal suite have polynomial time worst-case runtime complexity.

Sampath et al. [25] used the *concept lattice* to identify a reduced set of Web user sessions that provide the same URL coverage as the original set of collected user sessions. The proposed technique has exponential runtime in the worse-case due to concept lattice construction. Von Ronne [29] generalized the HGS algorithm such that every requirement must be satisfied multiple times before it is considered fully exercised, in order to minimize suites with respect to a new *probabilistic statement sensitivity coverage* (PSSC) criterion. Jones and Harrold [18] described two techniques for test suite minimization that are tailored to be used specifically with the *modified condition/decision coverage* (MC/DC) criterion. Harder et al. [10] developed a minimization approach that uses an operational abstraction, which is a formal specification for software derived from *actual* behavior. The idea is to keep the tests that *change* the operational abstraction and remove those tests that do not change the operational abstraction. Offutt et al. [20] presented an approach for reducing test suites by selecting test cases based on the additional requirement coverage by

considering tests in an order *different* from the order in which they were selected originally (such as in reverse order). Heimdahl and George [12] presented a heuristic for test suite minimization in the context of tests generated for specification-based criteria that are used for testing formal models of software. Black et al. [3] proposed a "bi-criteria" approach for test suite minimization that considers not only the coverage information for the tests, but also whether or not each test exposes a particular fault. This approach aims to compute optimally reduced suites containing the most fault-revealing tests.

A related topic is that of *test case prioritization*. In contrast to minimization techniques that attempt to remove test cases from a suite, prioritization techniques [24], [27] only reorder the execution of test cases within a suite with the goal of early detection of faults.

Empirical Studies. Rothermel et al. [22], [23] conducted a set of test suite minimization experiments with the Siemens suite [17]. Branch-coverage adequate suites were selected from the test pools and minimized using the HGS Algorithm [11]. On average, the test suites were reduced by 80 percent with 46 percent fault detection loss across all subject programs. Wong et al. [31] conducted experiments using test cases generated randomly, with suites created for various levels of nonadequate block coverage and optimally minimized using the ATACMIN tool [13] with respect to all-uses coverage. Suites were reduced in size up to 68 percent, and fault detection loss never exceeded 7 percent. Thus, compared to the results of Rothermel et al. [22], [23], Wong et al. showed that fault detection loss of minimized suites could be significantly less while at the same time achieving high suite size reduction.

Leon and Podgurski [19] conducted experiments to compare the results of test suite reduction and prioritization using the classical greedy heuristic with the distribution-based techniques that analyze the distribution of the *execution profiles* of tests. The results suggest that both approaches are complementary because they are each good at selecting tests exposing different types of faults.

Comparison of Prior Work with Our Approach. We have proposed a new approach that explicitly seeks to *include selective redundancy* in reduced suites with respect to a testing criterion. Our approach is general and can be integrated with a variety of existing *working list*-based test suite minimization techniques. For example, minimizing using an operational abstraction [10] can be extended using our approach as follows: A test case that does not change the operational abstraction may actually be retained if it is still considered important according to some *other* criteria (e.g., it covers a unique definition-use pair in the code). Also, the bi-criteria approach [3] could be extended as follows: A test that would normally be removed, because it does not expose the particular fault being considered, may still be retained if it exposes some *other* fault.

Our approach is also open to a variety of choices for additional criteria. For instance, we may combine our approach with the ideas presented by Heimdahl and George [12] to obtain testing requirements derived from formal specifications of software. We may also combine our approach with the ideas presented by von Ronne [29] to derive requirements that each need to be satisfied by *multiple* tests before they are considered sufficiently exercised. As another example, when analyzing the distribution of execution profiles [19], two tests whose execution profiles are in the same cluster, although only one test in a cluster might otherwise be retained, may still *both* be retained if, for example, those two tests have the greatest difference between each other among all tests in that cluster. Thus, the idea of our approach is relatively simple, yet also very versatile.

7 Conclusions

We have presented a new approach for test suite reduction that attempts to select those additional test cases that are redundant with respect to a particular coverage criterion, if the test cases are *not* redundant according to one or more other coverage criteria. This approach is based on the intuition that considering multiple testing criteria during test suite reduction is more effective than considering only one criterion, in terms of generating reduced suites with higher fault detection effectiveness. In our experimental study, our approach consistently performed better on average than other test suite minimization approaches by generating reduced test suites with less fault detection loss at the expense of only a relatively small increase in the sizes of the reduced suites. Our results suggest that the additional tests selected using our approach are those that are likely to expose additional faults in software.

Acknowledgments

The authors are grateful to Dr. Chandrasekhar Boyapati and Paul Darga, Department of Electrical Engineering and Computer Science, University of Michigan, for generating test cases for Java methods using the Korat tool for use in our experiments. The authors also thank Dr. Gregg Rothermel, Department of Computer Science, University of Nebraska, for providing the Siemens suite and the Space program, their instrumented versions, and the associated test case pools and faulty versions. Finally, they thank the anonymous referees for their thoughtful reviews that helped in significantly improving the initial version of the paper.

References

[1] H. Agrawal, "Efficient Coverage Testing Using Global Dominator Graphs," *Proc. Workshop Program Analysis for Software Tools and Eng.*, pp. 11-20, Sept. 1999.

[2] M. Balcer, W. Hasling, and T. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications," *Proc. Third Symp. Software Testing, Analysis, and Verification*, pp. 210-218, Dec. 1989.

[3] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction," *Proc. Int'l Conf. Software Eng.*, pp. 106-115, May 2004.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 123-133, July 2002.

[5] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135-141, Mar. 1996.

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

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. MIT Press, Sept. 2001.
[8] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, 1979.
[9] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering,* second ed. Prentice Hall, 2003.
[10] M. Harder, J. Mellen, and M.D. Ernst, "Improving Test Suites via Operational Abstraction," *Proc. Int'l Conf. Software Eng.,* pp. 60-71, May 2003.
[11] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, pp. 270-285, July 1993.
[12] M.P.E. Heimdahl and D. George, "Test-Suite Reduction for Model-Based Tests: Effects on Test Quality and Implications for Testing," *Proc. 19th IEEE Int'l Conf. Automated Software Eng.,* pp. 176-185, Sept. 2004.
[13] J.R. Horgan and S.A. London, "ATAC: A Data Flow Coverage Testing Tool for C," *Proc. Symp. Assessment of Quality Software Development Tools,* pp. 2-10, May 1992.
[14] http://www.cse.iitk.ac.in/users/jalote/download/javacoverage/index.html, 2005.
[15] http://www.cse.unl.edu/~galileo/sir, 2005.
[16] http://www.cs.fiu.edu/~weiss/dsaa_java/Code/DataStructures, 2005.
[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* pp. 191-200, May 1994.
[18] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 195-209, Mar. 2003.
[19] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 442-456, Nov. 2003.
[20] A.J. Offutt, J. Pan, and J.M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. Int'l Conf. Testing Computer Software,* pp. 111-123, June 1995.
[21] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367-375, Apr. 1985.
[22] 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," *Proc. Int'l Conf. Software Maintenance,* pp. 34-43, Nov. 1998.
[23] G. Rothermel, M.J. Harrold, J. von Ronne, and C. Hong, "Empirical Studies of Test-Suite Reduction," *Software Testing, Verification, and Reliability,* vol. 12, no. 4, pp. 219-249, Oct. 2002.
[24] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Oct. 2001.
[25] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A Scalable Approach to User-Session Based Testing of Web Applications through Concept Analysis," *Proc. 19th IEEE Int'l Conf. Automated Software Eng.,* pp. 132-141, Sept. 2004.
[26] G.W. Snedecor and W.G. Cochran, *Statistical Methods,* sixth ed. Iowa State Univ. Press, 1967.
[27] A. Srivastava and J. Thiagrajan, "Effectively Prioritizing Tests in Development Environment," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 97-106, July 2002.
[28] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *Proc. Workshop Program Analysis for Software Tools and Eng.,* Sept. 2005.
[29] J. von Ronne, "Test Suite Minimization: An Empirical Investigation," university honors college thesis, Oregon State Univ., June 1999.
[30] F.I. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenance,* pp. 44-53, Nov. 1998.
[31] 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.

Dennis Jeffrey received the BS and MS degrees in computer science from the University of Arizona in May 2003 and August 2005, respectively. He is a PhD student in the Department of Computer Science at the University of Arizona and expects to complete his PhD degree in computer science by August 2009. His research interests include software debugging, software testing, static and dynamic program analysis, and program slicing. He has published papers in the ICSM 2005 and COMPSAC 2006 conferences. His paper at COMPSAC 2006 received the Best Paper Award out of 183 submitted papers. He has been the recipient of several honors and awards, including the Outstanding Graduate Teaching Assistant Award from the Computer Science Department in May 2006 and the Galileo Circle Scholarship in 2003 and 2005, and he was on the Dean's List with Distinction every semester as an undergraduate student.

Neelam Gupta is an assistant professor of computer science at the University of Arizona. Her research areas include software testing, dynamic program analysis, and automated debugging. Her research has been funded by the US National Science Foundation, Microsoft, IBM, and the Arizona Center for Information Science and Technology (ACIST). She is a member of ACM and the IEEE. She has published papers in many prestigious conferences including ASE, PLDI, ICSE, FSE, FASE, ICSM, and COMPSAC. Her paper in COMPSAC 2006 won the Best Paper Award. Her paper in ASE 2001 was one of the five papers nominated for the Best Paper Award. She has served as a cochair of the program committee of the Fourth International Workshop on Dynamic Analysis (WODA 2006) and the program committee of the Third International Workshop on Software Quality Assurance (SOQUA 2006). She has also served or is serving on the program committees of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2007, ASE 2006, ASE 2003), the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2007), the International Computer Software and Applications Conference (COMPSAC 2007, COMPSAC 2006), the International Workshop on Dynamic Analysis (WODA 2004, WODA 2005), and the International Workshop on Security, Privacy, and Trust for Pervasive Applications (SPTPA 2006).