Test Case Prioritization Using Relevant Slices

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 and retesting occurs continuously during the software development lifecycle to detect errors as early as possible. The sizes of test suites grow as software evolves. Due to resource constraints, it is important to prioritize the execution of test cases so as to increase chances of early detection of faults. Prior techniques for test case prioritization are based on the total number of coverage requirements exercised by the test cases. In this paper, we present a new approach to prioritize test cases based on the coverage requirements present in the relevant slices *of the outputs of test cases. We present experimental results comparing the effectiveness of our prioritization approach with that of existing techniques that only account for total requirement coverage, in terms of ability to achieve high rate of fault detection. Our results present interesting insights into the effectiveness of using relevant slices for test case prioritization.*

1 Introduction

Software testing is an important and expensive stage of software development. As software changes over time, test suites are developed and used to test the modified software to make sure that changes do not affect the existing functionality in unintended ways, and to test for new functionality. This process is called *regression testing*. Due to time and resource constraints, it may not be possible to continually execute all the tests in suites on every testing iteration. It is therefore important to *prioritize* (order) the execution of test cases in test suites so as to execute those test cases early on during regression testing, whose output is more likely to change. Such a prioritization is expected to help in *early detection of faults* during regression testing. Techniques for *Test Case Prioritization* address this problem. In this paper, we present a new approach for prioritizing the execution of existing test cases with the goal of early detection of faults in the regression testing process.

The prior test case prioritization techniques studied in [2, 14] are primarily based on variations of the *total requirement coverage* and the *additional requirement coverage* of various structural elements in a program. For instance, total statement coverage prioritization orders test cases in decreasing order of the number of statements they exercise. Additional statement coverage prioritization orders test cases in decreasing order of the number of additional statements they exercise, that have not yet been covered by the tests earlier in the prioritized sequence. The prioritization techniques they do not take into consideration the statements or branches that actually influenced, or could potentially influence, the values of the program output. Neither do they take into consideration whether a test case traverses a modified statement or not while prioritizing the test cases.

It is intuitive to expect that the *output of a test case, that executes a larger number of statements that actually influence the output or have the potential to influence the output, is more likely to get affected by the modification than tests covering fewer such statements.* Additionally, tests exercising modified statements should have higher priority than tests that do not traverse any modifications. In this paper, we present a new approach for prioritizing test cases that is based not only on total statement (branch) coverage, but that also takes into account the number of statements (branches) executed that influence or have potential to influence the output produced by the test case. The set of statements that influence, or have potential to influence, the output of a program when run on a particular test case, correspond to the *relevant slice* [1, 10, 3] computed on the output of the program when executed by the test case. Our approach is based on the following observation. *If a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case.* Therefore, our heuristic for prioritizing test cases assigns higher weight to a test case with larger number of statements (branches) in its relevant slice of the output. We implemented our approach and

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

performed an experimental study using the programs from the Siemens suite [8] to evaluate the effectiveness of our approach in early detection of faults during regression testing. Our results show improvements over prior approaches based on total statement (branch) coverage and provide interesting insights into the use of size of relevant slices of outputs in determining the relative importance of test cases in revealing a change in the output during regression testing.

The remainder of this paper is organized as follows. The background for our work is explained in section 2. Our approach and its motivation are explained in Section 3. Our experimental study, along with results and discussion, are given in Section 4. Related work is discussed in Section 5, and our conclusions are given in Section 6.

2 Background

The test case prioritization problem can be defined [14] as follows.

Problem statement: Given a test suite T, the set PT consisting of all the permutations of test cases in T, and a function f from PT to the set of real numbers, find a $T' \in PT$ such that $(\forall T'')(T'' \neq T')[f(T') \geq f(T'')]$.

The specific focus of this paper and that of [14] is to find an ordering of test cases T' of a suite T with the goal of increasing the likelihood of revealing faults earlier in the testing process. A function f that quantifies the rate of fault detection for a test suite is the weighted "average percentage of faults detected" (APFD) value[1] computed during the execution of a test suite, as defined in [14]

The problem of test case prioritization as stated above is an optimization problem and has exponential complexity in the worst case. In [2, 14], the presented techniques are actually greedy heuristics in which test cases are ordered based upon their requirement coverage (such as the total number of statements or branches exercised). In their experimental studies in [2], the Siemens suite [8] programs were used to evaluate the effectiveness of the above heuristics. Each program in the Siemens suite is associated with a set of faulty versions (each containing a seeded error) and a pool of test cases. Thus, the effectiveness of the above test case prioritization heuristics in early execution of test cases that affect the output was measured by computing the rate at which the seeded faults (program modifications) were exposed by the test cases in the prioritized sequence. This rate was quantitatively measured by computing the APFD values for the prioritized test suites.

[1] In order to compute the APFD measure for the execution of a prioritized test suite, we plot the percentage of faults detected (y-axis) versus the fraction of test suite executed (x-axis). The area under the curve interpolating the points in this plot is the APFD measure. More area under the curve (a higher APFD value) indicates that more faults were detected when a smaller fraction of the prioritized suite was executed (more faults were detected earlier on during execution of tests in the suite)

We use the notion of a *relevant slice* [10, 1, 3] to identify the set of statements or branches that either influence, or have potential to influence, the value of the output produced by the test case.

Definition: Given a program P and its execution trace for a test case t, a statement s in the execution trace is in the *relevant slice* of the output if (1) at least one of the outputs produced by the execution of t is directly or indirectly data or control dependent upon s; or (2) s is either a predicate or a data-dependency of a predicate that may affect at least one output produced by the execution of t *if* the predicate evaluates to a *different* outcome.

Essentially, for a statement to be in the relevant slice, it must have either affected, or have potential to affect, [1, 3] the output produced by a test case.

3 Our Approach

Intuitively, if a test case exercises a modified statement, and that modified statement is also included in the relevant slice of the output, then most likely the output of the test case will be affected due to the modification. This is because every statement in the relevant either affects the output or has potential to affect the output of the test case. However, there can be modifications such as if a value is incremented and subsequently decremented in the code before the value is used again in which the modification may not affect the output. We illustrate this with the example program given in Figure 1 when it is executed with the input T = (a,b,c) = (1,5,4). The execution trace of the program for test T has branch B_1 evaluating to $false$ and branch B_3 subsequently evaluating to $true$, and therefore the program outputs the value 1.

```
1:    read(a, b, c);
2:    int x=0, y=0, z=0;
3:    x := a + 1;
4:    y := b + 1;
5:    z := c + 1;
6:    int w := 0;
B1:   if x > 3 then
B2:      if z > 4 then
7:          w := w + 1;
         endif
      endif
B3:   if y > 5 then
8:       w := w + 1;
      endif
9:    write(w);
```

Figure 1. An example program. Given input (a,b,c) = (1,5,4), the shaded lines indicate the relevant slice for output variable w at line 9; the darker-shaded lines distinguish those statements contained in the relevant slice that are *not* also contained in the dynamic slice.

Notice in the figure that the shaded lines (both the lightly-shaded and darker shaded lines) indicate the relevant slice of output variable w at line 9. The darker-shaded lines indicate those statements that are in the relevant slice, but *not* also in the dynamic slice, of variable w at line 9. The reason branch B_1 is not contained in the *dynamic* slice is because the value of w at line 9 is neither directly nor indirectly data or control-dependent on B_1 for test *T*. However, B_1 is contained in the *relevant* slice because although it did not affect the value of the output in this case, it could *potentially* affect the output *if* the branch outcome had evaluated to $true$ instead of $false$ (because the $true$ block of branch B_1 contains a potential definition of variable w). Similarly, since B_1 is data-dependent upon the definition of x in line 3 (but no other statements are data-dependent upon x defined in line 3), the line 3 is also contained in the relevant slice but not in the dynamic slice (ie., line 3 did not actually affect, but has potential to affect, the value of variable w at line 9).

Now suppose that the statement at line 3 gets modified by mistake so that it changes from `x:=a+1` into `x:=b+1`. Since line 3 is exercised and is also included in the relevant slice of output variable w at line 9, we intuitively expect that the error will be exposed by test case *T*. Indeed, with this modification it turns out that all three branches are exercised and evaluate to $true$. As a result, the value 2 is outputted (instead of the original value 1) and the erroneous modification is exposed. This example highlights the fact that the statement at line 3 certainly does have potential to affect the program output.

While we might generally expect that an exercised modification that is also included in the relevant slice will affect the output of a program, there do exist cases in which exercising a modification, that is contained in the relevant slice, will actually *not* change the output value(s) of a program and therefore may not lead to a fault being exposed. Consider the case in Figure 1 in which line 3 is again modified, but now it changes from `x:=a+1` into `y:=a+1`. With this modification, the execution actually follows the original path through the code in which B_1 evaluates to $false$ and B_3 subsequently evaluates to $true$, and therefore the original computed value for w, 1, is outputted. This demonstrates that it is not always true that exercising a modified statement, that has potential to influence program output, will actually cause the output value(s) to change. This implies that there is no guarantee that a test case exercising more modified statements that are contained in the relevant slice, will necessarily expose more faults (or change the output) than another test case exercising fewer modifications that are in the relevant slice. Nevertheless, in practice we might expect a higher likelihood that exercising a modification that has potential to affect program output, will actually cause program output to change and therefore expose a fault.

Let us next consider the case in which a modified statement is *not* contained in the relevant slice. From Figure 1, notice that line 5 is not contained in the relevant slice. This is because branch B_2 is the only statement in the program that is data-dependent upon the definition of z in line 5, but branch B_2 is not even exercised by our test *T* (thus, line 5 is not included in the relevant slice). Suppose that line 5 is changed from `z:=c+1` into `z:=b+1`. With this modification, the execution trace follows the original path of B_1 evaluating to $false$ and B_3 evaluating to $true$, and the output is not affected (since the modification affected the value of the definition which is not used in the relevant slice). A test case exercising a modified statement, that is not contained in the relevant slice, will likely change the program output *only when the variable being defined by the modified statement is changed to another variable which is used in the relevant slice* (e.g., by modifying the variable in the left-hand side of an assignment). To see this, suppose that line 5 is changed from `z:=c+1` into `y:=c+1`. Then this actually causes the executed path to change because now although B_1 still evaluates to $false$, we have that B_3 now subsequently evaluates to $false$ too. This causes the outputted value of w to be 0, which is different from the original outputted value of 1. However, we believe this case (in which the left-hand side of an assignment is modified) to be the only case in which an exercised modification will be *outside* the relevant slice, yet may still affect the program output.

The above example motivates the following approach. While considering whether a test case is likely to change the output when executed for the modified program, we need to take into consideration the following factors.

1. The number of statements (branches) in the relevant slice of output for the test case because any modification should *necessarily* affect some computation in the relevant slice to be able to change the output for this test case.

2. The number of statements that are executed by the test case but are not in the relevant slice of the output because changing the variable on lhs of an assignment not in the relevant slice may affect a computation in the relevant slice and thus may change the output.

Based on the above factors, we propose the following heuristic for test case prioritization. We order the test cases in decreasing order of test case weight, where the weight for a test is determined as follows: *test case weight = # of req's present in the relevant slice + total # of req's exercised by the test case*. Ties are broken arbitrarily. This criterion essentially gives "single" weight to those exercised requirements that are outside the relevant slice, and "double" weight to those exercised requirements that are contained in the relevant slice. We call this approach the

"REG+OI+POI" approach for prioritization, where REG, denotes REGular statement (branches) executed by the test case, OI denotes the Output Influencing and POI denotes the Potentially Output Influencing statements (branches) executed by the test case. In the next section, we present some experiments evaluating the effectiveness of the above heuristic for test case prioritization.

4 Experimental Study

4.1 Experimental Study

We used the Siemens programs described in Table 1 as our subject programs. Each program is associated with a set of faulty versions (each containing a seeded error) and a pool of test cases. The subject programs and their associated faulty versions and test pools were obtained from [7].

Program Name	Lines of Code	# of Faulty Versions	Test Case Pool Size	Program Description
tcas	138	41	1608	altitude separation
totinfo	346	23	1052	info accumulator
sched	299	8	2650	priority scheduler
sched2	297	10	2710	priority scheduler
ptok	402	7	4130	lexical analyzer
ptok2	483	10	4115	lexical analyzer
replace	516	32	5542	pattern substituter

Table 1. Siemens suite of programs.

The goal of our experiments is to see how well our prioritized suites for the heuristic proposed in section 3 perform in terms of rate of fault detection, with respect to the faulty versions provided with the Siemens programs. Our experimental approach is to generate test suites, prioritize them, and then measure the resulting APFD value for each prioritized suite. Similar to the experimental setup in [2, 14], we generated 1000 branch-coverage adequate test suites from the provided test pools for each program. For each test case, we measured the total set of exercised statements and branches using instrumented versions of the subject programs as generated by the Aristotle program analysis tool [6]. For each test case we also computed the set of statements and branches in the relevant slice based on the program output generated when executing the test case on the given subject program. To compute a relevant slice, we first computed the statements in the dynamic slice of the program output by computing the transitive closure of data and control dependences of the output exercised by the test case. Then, we augmented the dynamic slice with the additional statements/branches that were potentially-output-influencing (this would normally involve static analysis, but we tried to approximate this by looking at the execution traces of all the tests in the large test pools provided with the Siemens suite programs to see if changing the branch outcome would affect the program output or not) and their corresponding data dependencies to obtain the relevant slices.

Given the sets of regular exercised statements and branches along with the corresponding relevant slices for each test case, we used this information to prioritize the test cases and afterwards computed the APFD value for each prioritized suite in order to evaluate the suite's rate of fault detection[2]. 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.

We conducted experiments with the REG+OI+POI heuristic proposed in section 3. For comparison, we also prioritized the tests using the approach in [2, 14] that only accounts for total requirement coverage, and we ordered the test cases in decreasing order of the number of exercised statements/branches covered by each test case (ties are broken arbitrarily). We call this the "REG" approach.

4.2 Results and Discussion

The plots in Figures 2 and 3 illustrate the benefit of the REG+OI+POI approach over the REG approach in terms of promoting improved APFD values for prioritized suites. Specifically, for each plot, we consider every test suite for a given subject program in which the REG+OI+POI approach and the REG approach resulted in *different* computed APFD values. Then, for each such suite we compute the *difference* between the REG+OI+POI APFD value, and the REG APFD value, to obtain a measure of the improvement of the REG+OI+POI approach in terms of APFD value (negative differences occur when REG+OI+POI performs worse than REG). Finally, we order these suites in decreasing order of the amount of benefit of the REG+OI+POI approach, and then plot the data[3].

Table 2 is meant to accompany Figures 2 and 3. In this table, we present values computed for the area under the curve of each plot, including the areas both above and below the line $y = 0$. These areas respectively represent a measure of the improvement and the worsening witnessed

[2] To measure fault detection, similar to as done in [2, 14], we measured the set of faults exposed by each test case by executing every test case on each faulty version of the corresponding subject program, and compared the output of these faulty versions to the output generated when the test was run on the base version (the "oracle") of the program. If the base version output differed from some faulty version output when both versions were run on a particular test case, this indicated that a fault was exposed.

[3] In a graph, each unit along the x-axis represents a suite, and the y-axis represents the percentage difference in APFD values for a suite between the REG+OI+POI and the REG approaches. Each graph also includes the line $y = 0$, because the point at which the curve crosses this axis as one moves in the positive x direction, is the point at which suites transition from being better to being worse using the REG+OI+POI prioritization approach. Also, each graph includes a vertical line at the midpoint along the horizontal axis to show the median suite in the ordered sequence of suites along the x-axis.

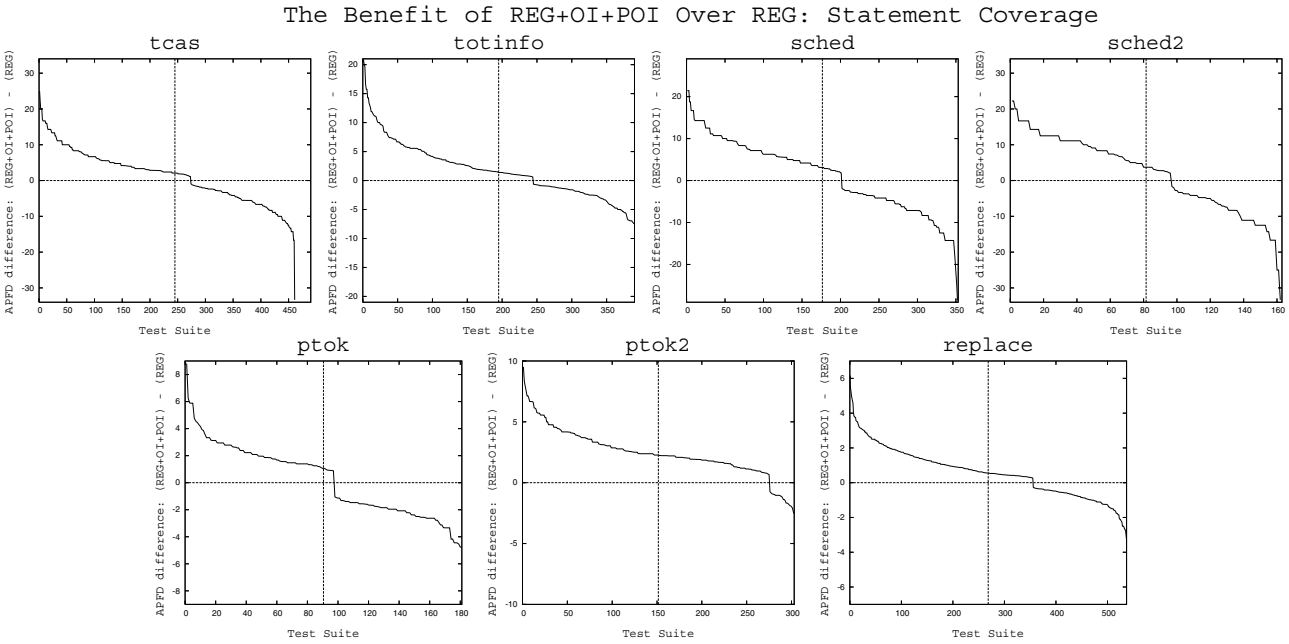

Figure 2. The difference between the APFD values of suites prioritized using REG+OI+POI, and the corresponding APFD values of the suites prioritized using REG, sorted by suite in the positive x direction in decreasing order of the amount of improvement of the REG+OI+POI approach (suites in which both REG+OI+POI and REG resulted in the same APFD value are not plotted). These plots are for prioritization when statement coverage is taken into account.

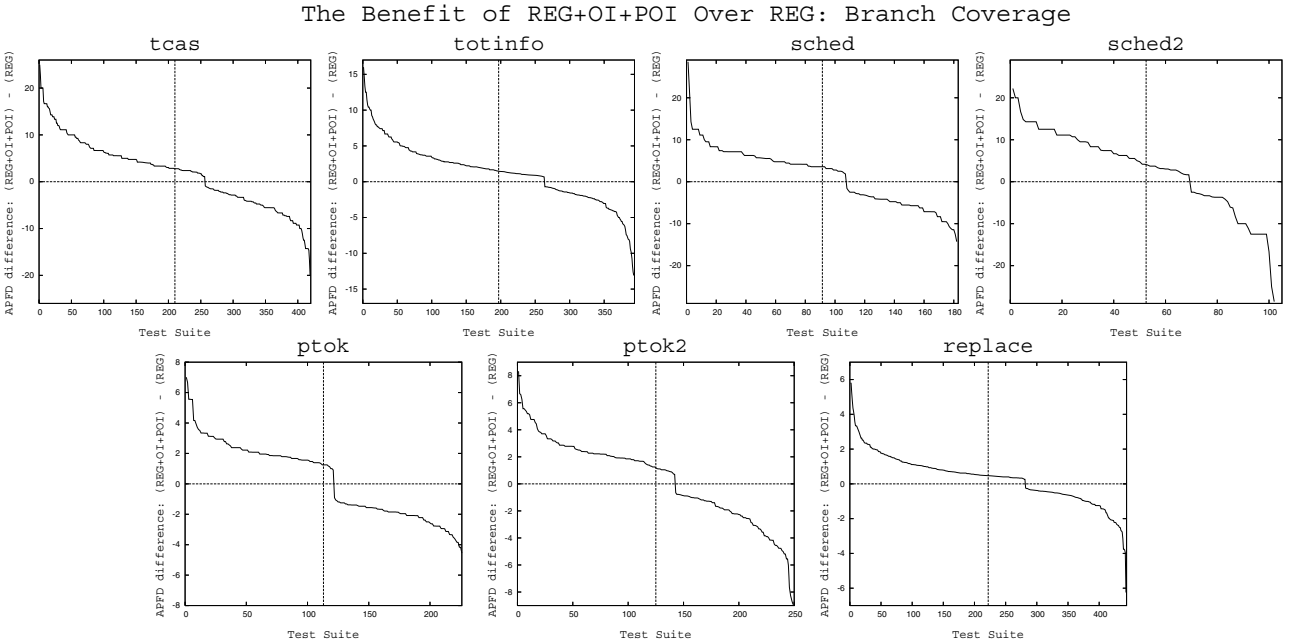

Figure 3. This figure is similar to Figure 2, except the plots here are for prioritization when branch coverage is taken into account.

Program Name	REG+OI+POI over REG: Area Under Curve to Accompany Figures 2 and 3					
	Statement			Branch		
	Area Above $y = 0$	Area Below $y = 0$	Ratio Above:Below	Area Above $y = 0$	Area Below $y = 0$	Ratio Above:Below
tcas	1689.66	1115.76	1.51	1668.80	888.79	1.88
totinfo	1091.33	556.82	1.96	915.39	405.73	2.26
sched	1510.27	1065.42	1.42	669.17	431.72	1.55
sched2	904.22	583.63	1.55	581.06	280.53	2.07
ptok	229.42	208.92	1.10	283.23	253.37	1.12
ptok2	782.71	100.72	7.77	363.96	284.72	1.28
replace	482.87	183.12	2.64	321.72	177.22	1.82

Table 2. Areas under the curves of the plots in Figures 2 and 3, both above the line $y = 0$ and below it, along with the corresponding ratio values.

when using REG+OI+POI instead of REG for prioritization. The area under each curve was computed in an approximate manner by summing up the areas of individual rectangles below each curve, where the width of each rectangle is one unit (one suite) and the height of each rectangle is the (absolute value of the) APFD difference for that suite. From the data in this table, it is clear that as Figures 2 and 3 suggest, the area under the curve *above* the line $y = 0$ is greater than the corresponding area under the curve *below* the line $y = 0$, in all cases. The columns of the Table 2 labeled "Ratio Above:Below" show the ratio of the area above $y = 0$ to the area below $y = 0$. In all cases, this ratio is greater than 1. Moreover, in 4 cases the ratio is greater than 2 (in one such case, the ratio is greater than 7), indicating that for these particular plots, there is more than twice the amount of area above the line $y = 0$ as below it. These results suggest that the cases in which REG+OI+POI leads to an improvement over REG may be much more significant than the cases in which REG+OI+POI leads to poorer results than REG.

In Table 3, we show the percentage of suites for each subject program such that the REG+OI+POI prioritization approach, resulted in better (higher), worse (lower), and the same APFD values as compared to the REG prioritization approach.

Program Name	REG+OI+POI over REG: % Suites Better/Worse/Same					
	Statement			Branch		
	Better	Worse	Same	Better	Worse	Same
tcas	27.3	18.8	53.9	25.6	16.3	58.1
totinfo	24.4	16.3	59.3	26.3	13.0	60.7
sched	20.1	15.1	64.8	10.7	7.5	81.8
sched2	9.6	6.6	83.8	6.9	3.3	89.8
ptok	9.7	8.7	81.6	12.1	11.0	76.9
ptok2	27.5	4.4	68.1	14.2	10.7	75.1
replace	35.5	18.4	46.1	28.1	16.2	55.7

Table 3. The percentage of suites for each subject program such that the REG+OI+POI approach yielded better, worse, and same APFD values as opposed to the REG approach, for both the statement and branch prioritizations.

From Table 3, we can see that in all cases, there are noticeably more suites that were improved by the REG+OI+POI approach than those suites that were made worse, in terms of APFD value. However, it is also the case that the majority of suites had the same APFD value across all programs when both the REG and REG+OI+POI approaches were used. The potential benefits of our approach are evident because significantly more suites are improved than made worse when output influences and potential output influences are taken into account during prioritization.

We also tried to experiment with a variant of the above approach. We grouped the test cases in a regression test suite into two sets, one set containing all those test cases that traversed a modification and the second set containing all the test cases that did not traverse a modified statement. We simply checked the statement number modified by the seeded fault in the program and then checked if the execution trace of the test case traversed that statement when it was executed for the correct version of the program. Note that if none of the statements covered by a test case are modified, it cannot change the output unless the program modifications introduce some new code on the path traversed by the test case. There were only 3 modifications in which new statements were added among all the seeded errors in the Siemens suite programs. So, after grouping the test cases in the above two sets, we put the set of test cases traversing modifications ahead of those not traversing modifications in the prioritized sequence. Next, we applied our REG+OI+POI heuristic to order the test cases within each set. We noticed significant improvement by this approach over the APFD values computed by REG only for two programs *ptok* and *ptok2*. Only for these two programs, the test suites on average contained between 2-10 and 2-4 test cases respectively that did not exercise any modification. For other programs, the modifications were such that almost all test cases in the test suites exercised the modifications and therefore the average APFD values did not change much with the above variant of the basic approach.

Although we obtained consistent improvement with our approach when compared with the REG approach, we did not observe very large improvements in the APFD values using our approach. We believe that the reason for this lies in the semantics of the programs in the Siemens suite and the small size of statement (branch) coverage adequate test suites for these programs. It turns out that for the Siemens subject programs, it is often *not* the case that traversing a modification will expose a fault in the software. Indeed, Table 4 shows that at best, only about a quarter of the cases

in which a modification is exercised actually leads to a fault being exposed. In some cases (especially `sched2`), the likelihood of exposing an error by traversing a modification is extremely small.

Program Name	% of Exercised Modifications that Lead to Exposed Faults
tcas	5.19
totinfo	19.44
sched	9.98
sched2	3.96
ptok	15.14
ptok2	26.76
replace	5.46

Table 4. For all test cases among the subject programs, the percentage of cases in which an exercised modification actually leads to a fault being exposed due to a change in program output.

The reason why many exercised modifications do not lead to a fault being exposed is evident by examining some of the particular modifications introduced in the subject programs. For example, in one case for `tcas`, a statement of the form $a = a > b$ has an operator change to become $a = a >= b$. However, this can only potentially cause the defined value for a to change when a happens to equal b. As another example for program `sched2`, there is a statement of the form $if(x = func())returnERROR$, which is changed to simply be $func()$. In this case, the output can only be affected in the cases when the if evaluates to $true$, which in turn may only occur in erroneous executions. As a final example with program `replace`, a statement of the form $if(x\&\&y)$ is changed to $if(x)$, which will not have an affect on the program execution if y happens to have value $true$. It turns out that many of the Siemens modifications are such that output can be affected only in certain cases, as in the above examples. The program `tcas` outputs only either an error message, or one of three possible output values. Programs `sched` and `sched2` simulate the execution of processes in a system, but the output consists only of the processes in the order in which they finish in the system. Program `replace` outputs the same data that was inputted, with the exception that all occurrences of a specified string are replaced with a new string. Given the specific output of these programs, it is understandable that output values may have less likelihood to be altered when a modified statement is executed (for example, perhaps a modification adjusts the priority of a process in the `sched` program, but the relative order in which processes finish in the system may remain unchanged). This observation explains why more significant improvements were not obtained by our approach when compared to the REG approach. Overall, our experimental results suggest that accounting for relevant slicing information, along with information about the modifications traversed by each test case, has potential when used as part of the test case prioritization process.

Based on our experimental results and analysis, we can also comment upon our expectations for how well REG+OI+POI prioritization may perform on programs that are much larger than the Siemens programs. On relatively large programs in which relatively few modifications are made in uncommonly-traversed sections of code, it would make more sense to follow the variant of our approach that first focuses on those test cases that traverse a modification, since many test cases may not traverse any modification and therefore cannot expose any additional faults. In this case, we expect that REG+OI+POI can lead to even greater improvements in rate of fault detection over REG, since REG does not consider which test cases actually traverse a modification. Also, large programs in which the output values are very sensitive to each individual computation are likely to show greater improvement with our approach as well, since one intuition guiding our heuristic is that traversing a modification will have an affect on the program output. On the other hand, large programs whose output values are not very sensitive to small changes in computation, or that have been modified in frequently-traversed sections of code, may not show as much improvement using REG+OI+POI, and may lead to results similar to those witnessed for the Siemens programs in our experiments. However, it is important to note that the degree of benefit of our approach will vary, for large or small programs, depending upon the program characteristics mentioned above and the distribution of modifications made to the programs.

5 Related Work

Several techniques [2, 9, 11, 14, 15] have been developed to prioritize the execution of existing test cases to expose faults early during the regression testing process. The technique developed in [15] prioritizes test cases in decreasing order of the number of impacted blocks (impacted blocks consist of old modified blocks and new blocks) that are likely to be covered by the tests; a heuristic is used to determine which impacted blocks are likely to be covered by each test. A test case prioritization technique that uses historical execution data was developed in [9]. In this approach, tests are prioritized based on how often a test has been run lately, how many faults the test has revealed recently, and the testing requirements have been exercised by the test case. The techniques discussed in [2, 11, 14] prioritize test cases based on variations of the total/additional requirement coverage information of the tests, as collected during previous testing of the software. In these studies, the prioritized test suites outperformed their unprioritized counterparts in terms of improving the rate of fault detection, but the relative performance of the prioritization techniques varied with the characteristics of test suites, programs, and faults. For example, in some cases total coverage techniques performed better than additional coverage techniques while the reverse was

true in other cases. Unlike the technique in [15], these techniques do not take the modifications made to the software into account while prioritizing the tests.

Korel and Laski [10] first introduced the notion of relevant slices. Agrawal et al. [1] presented an improved version of the definition of relevant slice and Gyimothy et al. [3] presented an algorithm for computing relevant slices. These authors suggest that looking at *potential* dependencies, besides just *actual* dependencies, are useful for debugging when modifications are made to software.

Our own work combines the concept of relevant slices [1, 3], and the coverage-based prioritization techniques presented in [2, 11, 14], to obtain a new approach for test case prioritization that accounts for the output-influencing and potentially-output-influencing coverage of test cases. As far as we know, the approach presented in this paper is the first attempt at incorporating the ideas of relevant slices in developing a new test case prioritization approach.

A related problem is that of *Test Case Selection* [4, 12, 1] for regression testing. A regression test selection technique chooses, from an existing regression test suite, a subset of test cases that are considered necessary to validate modified software. The approach in [4] uses static program slicing to detect definition-use associations that are affected by a program change. The work in [12] constructs control flow graphs for a procedure or program and its modified version, and uses these graphs to select test cases, from the regression test suite, that execute changed code. In [1], execution slice based, dynamic slice based and the relevant slice based approaches are proposed to determine the test cases in the regression test suite on which the new and old programs may produce different outputs.

The topic of *Test Suite Minimization* [13, 5, 17, 16] is related to that of test case prioritization because both share the common goal of reducing the cost of testing by providing testers with a way of identifying "important" test cases. However, unlike prioritization techniques, test suite minimization techniques permanently discard a subset of test cases from each suite. Prioritization techniques, on the other hand, only order all the test cases in a test suite without discarding any tests.

6 Conclusions

In this paper, we have presented a new approach for test case prioritization that takes into account the output-influencing and potentially-output-influencing statements and branches executed by the tests, as determined through the computation of relevant slices. We presented an experimental study comparing the effectiveness of our approach with a traditional prioritization approach that only accounts for the total (regular) statement and branch coverage of tests. Our experimental results show that our new prioritization approach is promising in terms of ordering the tests in suites so as to detect faults early in the testing process.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. *IEEE International Conference on Software Maintenance*, pages 348–357, 1993.

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

[3] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. *ACM/SIGSOFT Foundations of Software Engineering*, pages 303–321, 1999.

[4] R. Gupta, M. J. Harrold, and M. Soffa. An approach to regression testing using slicing. *IEEE International Conference on Software Maintenance*, pages 299–308, 1992.

[5] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[6] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. *Technical Report OSU-CISRC-3/97-TR17, Ohio State University*, March 1997.

[7] `http://www.cse.unl.edu/~galileo/sir`.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *IEEE International Conference on Software Engineering*, pages 191–200, 1994.

[9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. *IEEE International Conference on Software Engineering*, pages 119–129, 2002.

[10] B. Korel and J. Laski. Algorithmic software fault localization. *Annual Hawaii International Conference on System Sciences*, pages 246–252, 1991.

[11] G. Rothermel and S. Elbaum. Putting your best tests forward. *IEEE Software*, 20(5):74–77, Aug./Sept. 2003.

[12] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Software Engineering and Methodology*, pages 173–210, 1997.

[13] 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. *IEEE International Conference on Software Maintenance*, pages 34–43, 1998.

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

[15] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–106, 2002.

[16] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGPLAN SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Sept. 2005.

[17] 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.