Incremental Regression Testing*

Hiralal Agrawal
Joseph R. Horgan
Edward W. Krauser
Saul A. London

Bellcore
445 South Street
Morristown, NJ 07960
{*hira,jrh,ewk,saul*}*@bellcore.com*

Abstract

The purpose of regression testing is to ensure that bug fixes and new functionality introduced in a new version of a software do not adversely affect the correct functionality inherited from the previous version. This paper explores efficient methods of selecting small subsets of regression test sets that may be used to establish the same.

1 Introduction

Software inevitably changes however well conceived and well written it initially may be. Operational failures expose faults to be repaired. Mistaken and changed requirements cause the software to be reworked. New uses of old software yield new functionality not originally conceived in the requirements. The management of this change is critical to the continuing usefulness of the software.

The new functionality added to a system may be accommodated by the standard software development processes. Regression testing attempts to *revalidate* the old functionality inherited from the old version. The new version should behave exactly as the old except where new behavior is intended. Therefore, regression tests for a system may be viewed as partial operational requirements for new versions of the system.

Figure 1 shows a typical example of a sequence of time intervals during the life of a software system. Note that the regression testing intervals occupy a significant fraction of the system's lifetime. Unfortunately, complete regression testing can not always be accommodated during frequent modifications and updates of a system as it is often time consuming. This may result in the escape of costly, improper changes into the field. Clearly, omiting or arbitrarily reducing the regression testing interval is not an acceptable solution to the problem of software revalidation.

After a program has been modified, we must not only ensure that the modifications work correctly but also check that the unmodified parts of the program have not been adversely affected by the modifications. This is necessary because small changes in one part of a program may have subtle undesired effects in other seemingly unrelated parts of the program. Even though the modified program may yield correct outputs on test cases specifically designed to test the modifications, it may produce incorrect outputs on other test cases on which the original program produced correct outputs. Thus, during regression testing, the modified program is executed on all existing regression tests to verify that it still behaves the same way as the original program, except where change is expected.

In this paper, we propose some methods using which the test cases in the regression test suite whose outputs may be affected by the changes to the program may be identified automatically. Only these test cases need to be rerun during regression testing. Furthermore, the bulk of the cost of determining these tests is relegated to off-line processing, as depicted in Figure 2.

We refer to the problem of determining the test cases in a regression test suit on which the modified program may differ from the original program as the *incremental regression testing* problem. The solution to this problem requires that the answer be deter-

*This paper was published in the *Proceedings of the 1993 IEEE Conference on Software Maintenance*, Montreal, Canada, September 27–30, 1993.

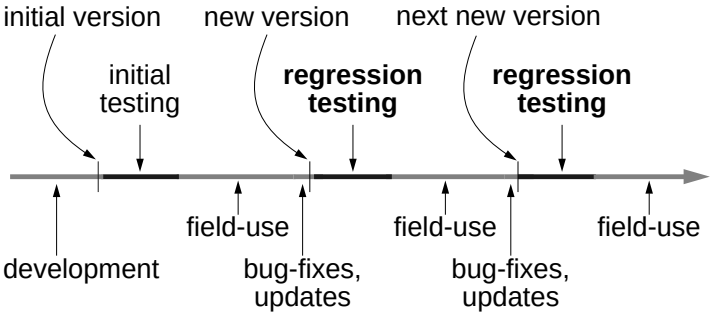

Figure 1: Regression Testing

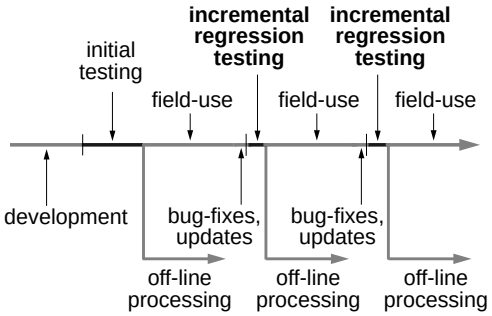

Figure 2: Incremental Regression Testing

mined solely on the basis of the analysis of the original program, the modifications, the regression test cases, and the original program's execution on these test cases—the modified program may not be executed on any test case. More precisely, the problem may be stated as follows:

> Given a program, P, its regression test set, T, and a "new" program version, P', find, T', the minimal subset of T necessary to revalidate the part of P''s functionality that is inherited from P.

In other words, if P and P' may be viewed as functions over sets of test cases, then determine the smallest subset, T', of T, such that:

$$P'(T') = P(T') \implies P'(T) = P(T)$$

Note that the effectiveness of T' in validating P' is limited by the effectiveness of T in validating P. If T performs an adequate task of validating P, T' will achieve the same for the corresponding functionality in P'. By the same token, if T validates only a fraction of P's functionality, so will T'. Also note that besides T', *new* test cases may be required to validate any new functionality in P' not inherited from P.

The incremental regression testing problem in its general formulation, as stated above, is an undecidable problem. In this paper, we present several techniques to obtain good approximate solutions to this problem.

2 Comparison with Related Work

There are two reasons for repeating existing tests on a program after it has been modified:

1. To verify that its behavior is unchanged except where required by the modifications.
2. To reevaluate the coverage achieved by the regression test suite using an appropriate coverage criterion, e.g., statement or data-flow coverage.

Most existing incremental regression testing techniques described in the literature [5, 8, 9, 10, 11, 17, 18, 19, 20] seem motivated by the second reason even though the first is of overwhelming practical significance[1]. All these techniques essentially address the following problem: How to achieve the required coverage of the modified program with minimal rework? They are not concerned with finding the test cases on which the original and the modified programs differ. Thus, they may select test cases that fail to distinguish the new program from the old for reexecution. Similarly, they may omit test cases on which the two programs differ from being selected. The goal of the techniques proposed in this paper, on the other hand, is to determine the test cases in the regression test suite on which the new and the old programs may produce different outputs. Only these test cases need to be rerun to establish that the new program preserves the desired functionality of the old program[2].

3 A Simple Model

A program may be changed for two reasons: (1) to fix faults, and (2) in response to changes in its specifications. Consider the former reason first. In this case, we must run the new program on all test cases on which the old program produced incorrect outputs to verify that the new program produces correct outputs on them. For the remaining test cases, the new program is expected to produce the same outputs as before. If we can determine a subset of these test cases where it may be asserted that the new program will produce the same output as before then we do not

[1] See [4] for a brief review of the cited techniques and a more detailed discussion on how they differ from the techniques presented here.

[2] Other test cases may have to be rerun to compute a coverage measure achieved by the regression test suite on the new program. Existing techniques cited above may be used to efficiently select such test cases.

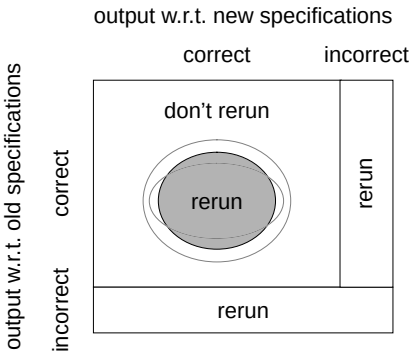

Figure 3: A simple model of the test cases to be rerun

need to execute the new program on them. We only need to execute it for the remaining test cases.

Now consider the case when a program is modified to accommodate changes in its specifications. In this case, we must rerun the new program on those test cases whose outputs, even though correct with respect to the old specifications, are considered incorrect with respect to the new specifications. For the remaining test cases, as in the previous case, there is no need to run the new program on a test case if it can be determined that the new program will produce the same output on it as before. Otherwise, the new program must be run on that test case.

Figure 3 depicts these cases. The top left rectangle in the figure represents test cases whose outputs are correct with respect to both the old and the new program specifications. The shaded oval inside the rectangle denotes the set of test cases on which the new program differs from the old. The new program must be executed on these test cases. There is, however, no method of deciding membership in this set without running the new program on all tests in the rectangle. The two dotted ovals represent approximations of that set given by our techniques. The number of test cases that lie in the shaded oval is usually much smaller compared to the number of test cases outside it. Thus our methods may help significantly cut down the time and cost of performing regression testing.

For the test cases on which the original program produces correct outputs, and the same outputs are also considered correct for the new program, the original program may be thought of as an oracle by which the new program's outputs may be validated. The new program's output on such a test case will be correct if it agrees with the original program's output. Otherwise, changes to the program have introduced a new fault in it. The methods proposed in this paper may be used to identify a subset of the regression test cases where the new program's output is guaranteed to be the same as that of the old program. There is no need to execute the new program on these test cases. For the remaining test cases, on the other hand, no such guarantees are to be had. Thus we must execute the new program on them and verify that their outputs match those of the old program.

4 Incremental Regression Testing

The techniques described in this paper are based on the following simple observations:

1. If a statement is not executed under a test case, it can not affect the program output for that test case.
2. Not all statements in the program are executed under all test cases.
3. Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case.
4. Every statement does not necessarily affect every part of the program output.

To verify some of these observations, we conducted an experiment to find out what fraction of the program's basic blocks are actually executed under each test case in a test suite used to validate the front-end modules of ATAC—a structural coverage testing tool developed at Bellcore [12, 13]. These modules contain about 14.5 thousand lines of C code, 6.5 thousand basic blocks, 183 functions, in 27 files. The test suite contains 413 test cases. We found that although these test cases together execute 71% of the basic blocks, an individual test case, on the average, executes only 11% of the blocks. We also discovered that although the test cases together exercise 88% of the functions, an individual test case in the suite, on the average, exercises only 26% of the functions. The techniques described in this paper exploit these observations.

Before we present the techniques, for the sake of simplicity in exposition, let us assume that the changes made to the program are such that the following two conditions hold:

- The control flow graph of the program remains unchanged, i.e., no existing edges are deleted and no new ones are introduced.
- The def-sets of nodes in the program's control flow graph remain unchanged, i.e., no changes are made to the left-hand-sides of assignment statements in the program.

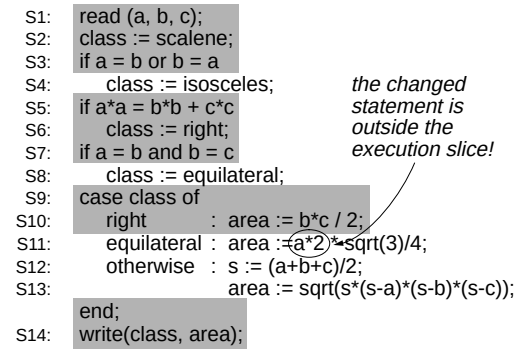

Figure 4: The Execution Slice with respect to T_3

Testcase	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	2.60
T_6	4	3	3	scalene	4.47

failures!

Figure 5: An example test suite

In Section 5, we will relax these assumptions and describe how the techniques that rely on them may be modified.

4.1 The Execution Slice Technique

Our first technique is based on Observations 1 and 2 above. Suppose the program modification consists of changing a single statement in the program. If, for a given test case, control never reaches that statement during the original program's execution, we can be sure that it will never reach it during the new program's execution either. The new and the old programs will agree in their outputs and we do not need to run the new program on that test case. This technique can obviously be generalized to the case when multiple changes are made: If a test case does not execute any of the modified statements, it need not be rerun.

We refer to the set of statements executed under a test case as the *execution slice* of the program with respect to that test case. Our first strategy, therefore, may be stated as follows:

> During the off-line processing depicted in Figure 2, find the execution slices of the program with respect to all test cases in the regression test suite. Then, after the program is modified, rerun the new program on only those test cases whose execution slices contain a modified statement.

Consider, for example, the program in Figure 4. It reads the lengths of the three sides of a triangle (assumed to be in descending order), classifies the triangle as one of a scalene, isosceles, right, or an equilateral triangle, computes its area using a formula based on the class, and finally, outputs the class and the area computed. Suppose it was initially tested using the first four test cases, T_1 through T_4, shown in Figure 5. The program produces correct outputs on all four test cases. Suppose T_5, also shown in Figure 5, is now added to the initial test suite. The program correctly classifies the triangle as an equilateral triangle for this test case but incorrectly computes its area as 2.6 instead of the correct value, 3.9. Debugging reveals that there is a fault in statement S11: It uses the expression $a * 2$ instead of $a * a$.

Suppose the fault is corrected. The new program must be rerun on T_5 to verify that it now produces the correct output. Should it also be rerun on T_1 through T_4? The shaded statements in Figure 4 denote the execution slice of the original program with respect to T_3. Note that S11 is not executed under this test case. As it is the only place where the new program differs from the old, the changed statement will not be reached when the new program is executed under T_3 either. Thus we can be sure that the new program will produce the same output on T_3 as the original program. This analysis also holds for T_2 and T_4. Hence there is no need to execute the new program on T_2, T_3, and T_4. Thus the regression testing in this case consists of executing the new program on just two test cases, T_1 and T_5, out of the first five test cases in Figure 5.

Now suppose the same program is executed on T_6, also shown in Figure 5. The program incorrectly classifies this triangle as a scalene triangle instead of classifying it as an isosceles triangle. Further debugging reveals another fault in the program: The second operand of the logical-or operation in the predicate in S3 is incorrectly specified as $b = a$ instead of $b = c$. After we correct this fault, we must run the new program on T_6 to verify that the fault has been fixed. Which other test cases should we rerun? The execution slice technique in this case would imply that it should be rerun on all test cases as S3 is executed un-

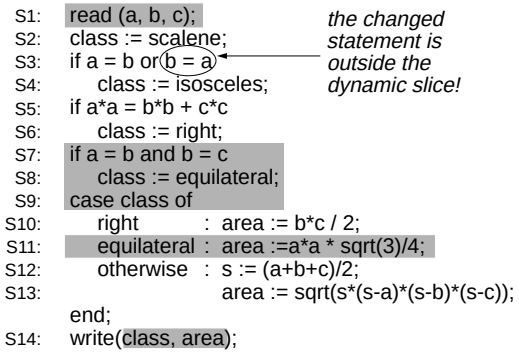

Figure 6: The Dynamic Program Slice for T_1

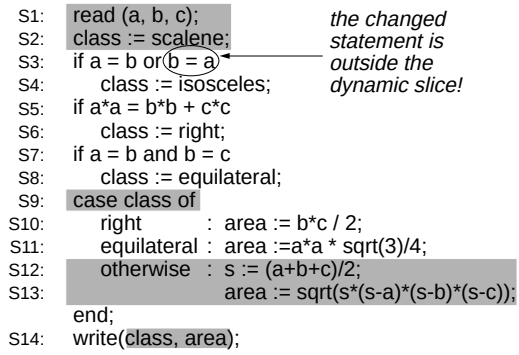

Figure 7: The Dynamic Program Slice for T_4

der all of them[3]. Clearly, applying the fix in this case will not affect the program output for the test cases that classify the triangle as an equilateral or a right triangle. In the next section we describe a technique that does not have this problem.

Instead of computing an execution slice at the statement level, we may also compute it at the function or the module level. That is, instead of determining which program statements are executed under each test case, we may simply determine which functions or modules are invoked under each test case. Then, we need to execute the new program on a test case only if a modified function or module was invoked during the original program's execution on that test case. This approach will give us a more conservative solution than that obtained using the statement level execution slices. But it also permits much lighter instrumentation making it more practical to use for large systems.

4.2 The Dynamic Slice Technique

Observation 3 suggests that not every statement that is executed under a test case has an effect on the program output for that test case. For example, although the predicate in S3 is executed under T_1 it does not affect the output under this test case. Even if it is modified so that it evaluates differently, it will not make any difference because the value of 'class' in this case is overwritten by S8.

If there was a way to determine, for each test case, the statements that have an effect on the program output, then we could easily identify the relevant test cases to be rerun. Dynamic program slices provide us exactly this functionality [3]. The dynamic program slice with respect to the output variables gives us the statements that are not only executed but also have an effect on the program output under that test case. A dynamic program slice is obtained by recursively traversing the data and control dependence edges in the dynamic dependence graph of the program for the given test case [1, 3].

Our second strategy, therefore, may be stated as follows:

> During the off-line processing depicted in Figure 2, find the dynamic program slices with respect to the program output for all test cases in the regression test suite. Then, after the program is modified, rerun the new program on only those test cases whose dynamic program slices contain a modified statement.

Figure 6 shows the dynamic program slice with respect to the program output for T_1. Note that it does not include the modified statement, S3. Thus, we need not run the new program on this test case. The dynamic slice technique requires that besides T_6 the new program be executed on just one more test case, T_2. The dynamic program slices with respect to other test cases do not include the modified statement.

Note that the above strategy only requires that we determine whether or not a modified statement belongs to a dynamic program slice. The nature of the modification does not matter. Figure 7 shows the dynamic program slice with respect to T_4. As the change lies outside the slice, the above strategy implies that the new program need not be rerun on T_4. The change here, as mentioned earlier, consisted of replacing the expression $b = a$ by $b = c$ in S3. If we execute the new program on T_4, it will, as expected, produce the same output as the old program. But what if we had erroneously changed the faulty expression to $b = b$ instead of $b = c$? The dynamic slice strategy would still imply that the new program need not be rerun on

[3] For the purposes of this example assume that short circuit evaluation of a predicate, as in C, is not permitted.

T_4 because it only depends on which statements were changed, not on how they were changed. The new program in this case, however, will produce a different output on T_4 compared to the original program. Thus the dynamic slice technique will fail to identify T_4 as a relevant test case to be rerun in this case. In the next section, we present a technique that does not have this problem.

4.3 The Relevant Slice Technique

A dynamic program slice with respect to the program output includes only those statements that were executed and actually had an effect on the output. Besides such statements, we also need to identify those statements that were executed and did not affect the output, but could have affected it had they evaluated differently. We need to find such statements because if any changes are made to them they may evaluate differently and change the program output.

Korel and Laski have defined a notion of "potential influence" in the context of fault localization that is very close to the functionality we described above [16]. According to that definition, a use of a variable, v, at a location, l, in a given execution history is said to be potentially dependent on an earlier occurrence, p, of a predicate in the execution history if (1) v is never defined between p and l but there exists another path from p to l along which v is defined, and (2) l is not control dependent on p. Thus the use of 'class' in S9 in Figure 7 is potentially dependent on the predicates in S3, S5, and S7, because each of them could have potentially caused other paths to be traversed along which 'class' is defined, thereby potentially affecting its value at S9. Unfortunately, this definition may also cause some unnecessary predicates to be identified as potentially affecting a use of a variable. For example, consider the following code segment:

```
1:  x = ...;
2:  if (C1) {
3:      if (C2)
4:          x = ...;
5:      y = ... x ...;
    }
6:  z = ... x ...;
```

Suppose C1 evaluates to true and C2 evaluates to false, i.e., the execution history of the above segment consists of the path <1,2,3,5,6>. The use of 'x' on line 5 in this case is potentially dependent on C2 but not on C1 as line 5 is control dependent on C1. The use of 'x' on line 6 is similarly potentially dependent on C2. But, according to the above definition, it is also potentially dependent on C1 as there exists another path, <2,3,4,5,6>, along which 'x' is defined and line 6 is not control dependent on C1. But, in this case, even if C1 had evaluated to false instead of true it would not have affected the value of 'x' on line 6. Therefore, we would not want the use of 'x' on line 6 in this case to be potentially dependent on C1. We would only want it to be potentially dependent on C2 because if C2 had evaluated to true instead of false it would have affected the value of 'x' on line 6.

```
static_defs = static reaching definitions of var at loc;
dynamic_def = the dynamic reaching definition of var at loc;
control_nodes = the closure of the static control dependences
                    of the statements in static_defs;
initialize potential_deps to null;
mark all nodes in the dynamic dependence graph between loc
        and dynamic_def as unvisited;
for each node, n, in the dynamic dependence graph starting at
        loc and going back up to dynamic_def do
    if n is marked as visited then
        continue; /* i.e., skip this node */
    mark n as visited;
    if n belongs to control_nodes then
        add n to potential_deps;
        mark all the dynamic control dependences of n
                between n and dynamic_def as visited;
    end-if;
end-for;
return potential_deps;
```

Figure 8: An algorithm to compute the potential dependences of a variable, *var*, at a location, *loc*, in a given execution history.

The following revised definition of potential dependence does not have the above problem. The use of a variable, v, at a location, l, in a given execution history is said to be *potentially dependent* on an earlier occurrence, p, of a predicate in the execution history if (1) v is never defined between p and l but there exists another path from p to l along which v is defined, and (2) *changing* the evaluation of p may cause this untraversed path to be traversed.

According to this definition, in the above example the use of 'x' on line 6 is potentially dependent on C2 but not on C1 as changing the evaluation of C1 from true to false cannot cause the unexecuted assignment of 'x' on line 4 to be executed. Figure 8 shows an algorithm to compute potential dependences of a variable use for a given execution history.

If we include a predicate in a slice because it may potentially affect the program output, we must also include the statements that in turn affect the included predicate in the slice. But we should only include the data and potential dependences of this predicate, not its control dependences. Including the control dependences will nullify the very reasons for revising the

definition of potential dependence above. For example, doing so will require that C1 be also included in the slice with respect to 'x' on line 6 in the example above because C2 is control dependent on C1 even though changing C1 will not affect the value of 'x' in this case.

We refer to the set of statements that, if modified, may alter the program output for a given test case as the *relevant slice* with respect to the program output for that test case. Remember that the modifications should be such that they do not violate the two assumptions made in Section 4 about them (these assumptions are relaxed in Section 5). The dynamic program slice with respect to the program output for a given test case gives the statements that were executed and had an effect on the program output. If a change is made to any of these statements it may obviously affect the program output. To find the other statements which, when modified, may alter the program output, we need to identify the predicates on which the statements in the dynamic slice are potentially dependent, as well as the closure of data and potential dependences of these predicates. This can be easily achieved by adding another set of edges to the dynamic dependence graph [1, 3] denoting the potential dependences, and finding all nodes that may be reached from any node in the dynamic program slice by following just the data and the potential dependence edges. The set of the statements that correspond to the nodes selected above, including those in the dynamic program slice, gives us the desired relevant slice.

Our third strategy, therefore, may be stated as follows:

> During the off-line processing depicted in Figure 2, find the relevant program slices with respect to the program output for all test cases in the regression test suite. Then, after the program is modified, rerun the new program on only those test cases whose relevant slices contain a modified statement.

Figure 9 shows the relevant slice for T_4. Note that the modified predicate in S3 belongs to the relevant slice. Thus the above technique will require that the new program also be rerun on T_4. Besides T_4, this strategy will require that the new program be rerun on the two test cases identified by the dynamic slice technique, T_2 and T_6. The relevant slices for these two test cases are the same as that in Figure 9, except that S4 is included in their relevant slices instead of S2. Of the six test cases shown in Figure 5, the change in S3 may affect the program output only for T_2, T_4, and T_6. The relevant slice technique identifies exactly these three test cases to be rerun.

```
S1:  read (a, b, c);                          the changed
S2:  class := scalene;                        statement
S3:  if a = b or b = a                        belongs to the
S4:      class := isosceles;                  relevant slice!
S5:  if a*a = b*b + c*c
S6:      class := right;
S7:  if a = b and b = c
S8:      class := equilateral;
S9:  case class of
S10:     right      : area := b*c / 2;
S11:     equilateral : area :=a*a * sqrt(3)/4;
S12:     otherwise  : s := (a+b+c)/2;
S13:                  area := sqrt(s*(s-a)*(s-b)*(s-c));
     end;
S14: write(class, area);
```

Figure 9: The Relevant Program Slice for T_4

Note that the algorithm to compute potential dependences shown in Figure 8 requires, among other things, the computation of *static* data dependences. Unfortunately, precise static data dependences are hard to compute when the program makes use of pointers, arrays, and dynamic memory allocations. One must make conservative assumptions while computing reaching definitions in the presence of such features. For example, in the presence of unconstrained pointers, as in C, one must, in general, assume that an indirect assignment via a pointer may conceivably modify any variable. We do not need to make such conservative assumptions while computing the *dynamic data* dependences because when executing the program on a given test case, we know the precise memory locations pointed to by pointer variables. Therefore, we may obtain precise dynamic program slices even in the presence of unconstrained pointers [1]. We may not obtain precise relevant slices as their computation depends on imprecise static data dependence information.

It may be pointed out that it is not just the algorithm in Figure 8 that requires static information to compute potential dependences. Any other algorithm to compute potential dependences would require the same static information, because, by definition, potential dependence requires searching for paths that were *not executed* but could have affected the program output had they been executed.

4.4 Other Techniques

The difference between a dynamic program slice and the corresponding relevant slice is that the latter also includes certain predicates on which statements included in the former are potentially dependent (as

well as certain dependences of those predicates). Thus the predicates included in the relevant slice form a superset of those included in the dynamic slice. In the presence of indirect references via pointers, all predicates that have been executed but not included in the dynamic slice that also control an unexecuted assignment via a pointer would tend to be included in the relevant slice. If the program contains many indirect references, the difference between the set of executed predicates and the set of predicates included in the relevant slice will tend to reduce. Thus, a simpler but more conservative approach would be to add all predicates that were executed to the slice being constructed. Adding these predicates would require that we also recursively add their data dependences to the slice. We refer to the resulting slice as an *approximate relevant slice*. It is a superset of the corresponding relevant slice but a subset of the corresponding execution slice. The advantage of using an approximate relevant slice over a relevant slice is that the former does not require the computation of potential dependences that may be imprecise anyway.

Observation 4 suggests that not every program statement necessarily affects every output variable. For example, in Figure 4, S11 may affect the value of 'area' but not that of 'class.' This observation may be used to further reduce the number of test cases that need to be rerun. We may determine the subset of the output variables that may be affected by the modifications by examining the *static* program slice [14, 21] with respect to each output variable and checking if it contains a modified statement. Also, during the offline processing, instead of finding the combined relevant slice with respect to all output variables, we may find the relevant slices with respect to individual output variables. Then, in order to decide if a test case should be rerun after the program is modified, we may check if the relevant slice with respect to an affected output variable contains a modified statement instead of checking if the combined relevant slice contains the same.

Figure 10 shows the inclusion relationships among various types of slices. The inclusion relationships among the sets of test cases selected to be rerun by the techniques based on these slices mirror the same relationships.

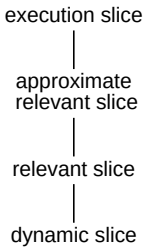

Figure 10: Inclusion relationships among various types of slices.

5 Relaxing the Assumptions

In Section 4 we assumed that no statements are added to or deleted from the program and no changes are made to the left-hand-sides of assignments. In other words, changes were confined to modifying the expressions in predicates and the right-hand-sides of assignments. Note that the execution slice technique does not rely on these assumptions at all. Thus it will also work after these assumptions are relaxed. In this section, we discuss how the relevant slice technique may be modified to work in the absence of these assumptions. The dynamic slice and other techniques may be adapted similarly.

Let us first consider the case of deletion of an assignment, $x = \ldots$. The effect of this deletion is the same as though the right-hand-side of the assignment had been "changed" to x. Therefore, the relevant slice technique in this case would be to rerun the new program on any test case whose relevant slice contains the deleted assignment.

Now consider the case where an assignment, $x = exp$, has been added to the program. Let $u_1, u_2, \ldots, u_n$ be the corresponding reaching uses of x in the new program. The effect of adding the new assignment would be the same as though the uses of x in all u_i's had been replaced by exp. Thus the addition may be treated as though all u_i's had been "changed". Therefore, the relevant slice technique in this case would be to rerun the new program on any test case whose relevant slice contains a u_i.

Now suppose an assignment, $x = exp$, in the program has been changed to $y = exp$. The effect of this change is the same as though the original assignment had been deleted and the new assignment had been added to the program. Thus the relevant slice technique in this case would be to rerun the new program on test cases that need to be rerun under any of these two changes.

Deleting a predicate has the same effect as though that predicate had been replaced by the boolean constant, $true$. Therefore, the relevant slice technique in this case would be to rerun the new program on a test case if its relevant slice contains the deleted predicate.

Next, consider the case where a predicate, $pred$, has been added to the program at a location, loc. Let

$s_1, s_2, \ldots, s_n$ be the statements that are directly control dependent on *pred* in the new program. Before the predicate was inserted, all s_i's were always executed whenever the control reached *loc*. But the new predicate may prevent some or all of the s_i's from being executed depending on its value. Therefore, the relevant slice technique in this case would be rerun the new program on any test case whose relevant slice contains an s_i.

Moving a statement from one location to another may be treated as if the statement had been deleted from its original location and inserted at its new location. Thus, in this case we should rerun the new program on all test cases that qualify to be rerun under either of these two changes.

6 Other Applications

The techniques presented in this paper also have other applications besides incremental regression testing. We briefly discuss two of them here: debugging and mutation testing.

Debugging a program that produces an incorrect output on a test case involves searching the program for faulty statements which, when corrected, will cause it to produce different, correct output. The techniques described in this paper do just that—identify statements which when modified may cause the new program to produce a different output on a given test case. Thus, the same techniques may also be used to expedite debugging. In fact, dynamic program slicing [3, 15], and the notion of potential dependence [16] used to define relevant slices in this paper, were first proposed in the context of debugging.

Mutation testing requires that a program be tested on enough test cases that can "kill" all or a desired fraction of the nonequivalent mutants of the program [6, 7]. A mutant is obtained by applying a simple syntactic change to the program, e.g., replacing a '<' operator in a predicate by a '>' operator, replacing a variable reference by another compatible variable, etc. A test case is said to *kill* a mutant if the mutant produces a different output on that test case compared to that produced by the original program. The techniques presented in this paper may be used to determine if a test case might kill a mutant without executing the mutant on that test case: If the mutated expression lies outside the relevant slice with respect to a test case then it may not kill that mutant. Thus we need not execute it on this test case.

7 Concluding Remarks

We have two tools—Spyder, that provides dynamic slicing facilities [2], and Atac, that provides structural coverage-based testing facilities [13]; both work on C programs. We are modifying Spyder so it may also compute relevant slices. Atac already indirectly provides facilities to obtain execution slices. We plan to experiment with the techniques proposed in this paper using these tools to empirically determine the differences in the numbers of test cases identified to be rerun by these techniques.

The amount of regression testing effort saved using the techniques discussed here obviously depends on the nature of test cases in the regression test suite as well as the extent and the locations of the changes made. If the test cases are numerous and they each exercise small parts of the program's functionality then using these techniques should lead to greater savings. If, on the other hand, we only have a few test cases and each of them exercises most of the program's functionality then our methods will be less useful. The program locations where changes are made may also have a major effect on the amount of savings implied by using these techniques. A single change to an initialization statement that affects the program output for almost all test cases means almost all test cases must be rerun. On the other hand, even if changes are made to many parts of the program that are rarely executed by the regression tests, our techniques may mean significant savings.

Many large applications may be intolerant of the intrusive instrumentation required by most of the techniques described in this paper, particularly during the system testing stage. The function or module level execution slice technique may be the only practical technique to use in these cases, as the instrumentation required in this case is fairly light weight. The remaining techniques, however, may be useful during unit testing. Experimentation using large programs and test suites will give us a clearer view of the relative usefulness of various techniques proposed here.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 60–73. ACM Press, Oct. 1991.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtrack-

ing. *Software Practice and Experience*, 23(6):589–616, June 1993.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*. ACM Press, June 1990. SIGPLAN Notices, 25(6):246–256, June 1990.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. L. London. Incremental regression testing. Internal memorandum, Bell Communications Research, Morristown, New Jersey, Dec. 1992.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396. ACM Press, Jan. 1993.

[6] R. A. DeMillo, D. S. Guindi, K. S. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra mutation system. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151. ACM Press, July 1988.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[8] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference*, pages B6.3.1–B6.3.6. IEEE Computer Society Press, 1981.

[9] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

[10] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 362–367. IEEE Computer Society Press, 1988.

[11] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, pages 31–36, Jan. 1990.

[12] J. R. Horgan and S. L. London. Data flow coverage and the C language. In *Proceedings of the Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 87–97. ACM Press, Oct. 1991.

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

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[15] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, Nov. 1990.

[16] B. Korel and J. Laski. Algorithmic software fault localization. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume II, pages 246–252, 1991.

[17] H. K. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69. IEEE Computer Society Press, 1989.

[18] T. J. Ostrand and E. J. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247. Lawrence and Craig, Sept. 1988.

[19] B. Sherlund and B. Korel. Modification oriented regression testing. In *Proceedings of the Fourth International Software Quality Week, San Francisco, CA, May 14-17, 1991*, 1991.

[20] A.-B. Taha, S. M. Thebaut, and S.-S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the Thirteenth Annal International Computer Software and Applications Conference (COMPSAC 89)*, pages 527–534. IEEE Computer Society Press, 1989.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.