A Study of Effective Regression Testing in Practice[*]

W. Eric Wong, J. R. Horgan, Saul London, Hira Agrawal
Bell Communications Research
445 South Street
Morristown, NJ 07960

Abstract

The purpose of regression testing is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change. Regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the software. Many techniques have been reported on how to select regression tests so that the number of test cases does not grow too large as the software evolves. Our proposed hybrid technique combines modification, minimization and prioritization-based selection using a list of source code changes and the execution traces from test cases run on previous versions. This technique seeks to identify a representative subset of all test cases that may result in different output behavior on the new software version. We report our experience with a tool called ATAC which implements this technique.

Keywords: *Regression Testing, Modification-Based Test Selection, Test Set Minimization, Test Set Prioritization*

1 Introduction

No matter how well conceived and tested before being released, software will eventually have to be modified in order to fix bugs or respond to changes in user specifications. Regression testing must be conducted to confirm that recent program changes have not adversely affected existing features and new tests must be conducted to test new features. Testers might rerun all test cases generated at earlier stages to ensure that the program behaves as expected. However, as a program evolves the regression test set grows larger, *old* tests are rarely discarded, and the expense of regression testing grows. Repeating all previous test cases in regression testing after each minor software revision or patch is often impossible due to the pressure of time and budget constraints. On the other hand, for software revalidation, arbitrarily omitting test cases used in regression testing is risky. In this paper, we investigate methods to select small subsets of effective fault-revealing regression test cases to revalidate software.

Many techniques have been reported in the literature on how to select regression tests for program revalidation. The goal of some studies [1, 3, 13, 21] is to select every test case on which the new and the old programs produce different outputs, but ignore the coverage of these tests in the modified program. In general, however, this is a difficult, sometimes undecidable, problem. Others [5, 8, 10, 15, 18, 20] place an emphasis on selecting existing test cases to cover *modified* program components and those may be affected by the modifications, i.e., they use coverage information to guide test selection. They are not concerned with finding test cases on which the original and the modified programs differ. Consequently, these techniques may fail to select existing tests that expose faults in the modified program. They may also include test cases that do not distinguish the new program from the old for reexecution.

In this paper, a combination of both techniques described above is used. We first select tests from the regression suite that execute any of the modifications in the old program and refer to this technique as a modification-based test selection technique. This includes tests that have to be used for revalidation, but it also contains some *redundant* tests on which the old and the new program produce the same outputs. Then, depending on the available resources, a *trade-off* between what we should ideally do in regression testing and what we can afford to do is applied to determine which tests, among those necessary, should be reexecuted first, and which ones have lower priority or are to be omitted from reexecution. Two techniques, test set minimization and prioritization, are used. Although both of them may exclude certain necessary regression tests, the value of using them is explained below.

[*]This paper was published in the proceedings of the 8th IEEE International Symposium on Software Reliability Engineering (ISSRE'97), pp 264-274, Albuquerque, NM, November 1997.

1. The significance of minimization is that the resulting minimized set has the same coverage with respect to a certain criterion (say $\mathcal{C}$) as the original set. This implies if a program attribute defined by $\mathcal{C}$ is covered by the original set, it will also be covered by the minimized set. We do minimization only on a subset of regression tests determined by using the modification-based test selection technique. The advantages are (1) reducing the amount of work required by test set minimization, (2) having a higher chance to select more tests on which the new and the old programs produce different outputs, and (3) having a lesser chance to include test cases that fail to distinguish the new program from the old.

2. The prioritization approach sorts regression tests selected using a modification-based technique in increasing order of additional coverage per cost. It is especially useful when testers can only afford to reexecute a few regression tests. Under this scenario, testers can select those tests starting from the top of the prioritized list. Such flexibility does not appear in any of the existing selective retest techniques.

The remainder of this paper is organized as follows. Section 2 describes the techniques and the tool used to select effective regression tests. A case study using these techniques is reported in Section 3. The impact of the our results in software regression testing is discussed in Section 4.

2 Method

In this section we explain the techniques and the ATAC tool used to select regression tests.

2.1 Modification-based test selection

The objective of regression testing is to test a modified program to gain confidence that recent changes have not adversely affected existing features. This can be achieved by rerunning only those test cases in the regression test suite on which the new and the old programs produce different outputs. Thus, given a program P, its regression test set T and the modified program P', we need to find $T' \subseteq T$ such that[1]

$$\forall\, t \in T,\ t \in T' \iff P'(t) \neq P(t) \tag{1}$$

[1] For simplicity of exposition, we assume no obsolete test cases in the regression test suite. Otherwise, such tests should be excluded before any tests are selected. The obsolete tests are those which no longer perform the testing functionality as they were designed to do because of requirement changes in the current version. We also assume the same testing environment other than code is used for testing P and P'. The notations "$\iff$", "$\forall$", and "$\in$" represent "if and only if", "for all", and "in", respectively.

In general, computing T' is not decidable for an arbitrary P, P' and T. In practice, one can find T' only by executing P' on every regression test case, which is what we want to avoid. However, instead of constructing T' from T, we can find T'', $T' \subseteq T''$, by including all tests in T which execute some modified code, i.e.,

$$\begin{aligned} &\forall\, t \in T,\ t \in T'' \iff \\ &\qquad t \text{ executes code in } P \text{ that was changed or deleted} \\ &\qquad \text{to form } P' \text{ or code in } P \text{ at which point new code is} \\ &\qquad \text{added to form } P' \end{aligned} \tag{2}$$

For a given test case t, if the code executed in P by t is the same as that in P', there will be no difference between the outputs generated from P and P' on t. This implies if $P(t) \neq P'(t)$, t must execute some code that was modified, with respect to P, for P', i.e., $t \in T''$. On the other hand, since not every execution of the modified code will affect the output for that test case, there may exist some $t \in T''$ such that $P(t) = P'(t)$. Therefore, T'' contains T' and can be used as a conservative alternative for T'. Hereafter, we refer to T'' as a modification-based subset of T. A similar approach is termed the "execution slice technique" by Agrawal *et al.* [1] and "modification-traversing tests" by Rothermel [17].

The solution to modification-based selection can be determined solely on the basis of the analysis of the original program, the regression test set, how the original program is executed by each regression test case, and the modifications. There is no need to execute the modified program on any of the regression test cases. In addition, as depicted in Figure 1, the execution information of each regression test case on the original program can be collected as soon as P is available. This relegates the bulk of the cost of selecting these tests to off-line processing.

2.2 Additional test selection by minimization or prioritization

Although T'' is *complete* in that it selects every test case from the regression testing suite that should be used for revalidating the functionalities inherited from the previous version, it is not *precise* because it includes tests on which P and P' produce the same outputs. One can of course use sophisticated techniques such as relevant slicing [1] and others proposed by different researchers as discussed in Section 1 to improve the precision of T''. However, these techniques are either expensive to use, or only supported by tools working under certain constraints.

Rather than overemphasize the importance of the selected subset of regression tests being absolutely *complete* and *precise*, we stress the *flexibility* of test selection. This is especially important when such a subset, even though perfect from the complete and precise point of view, is still

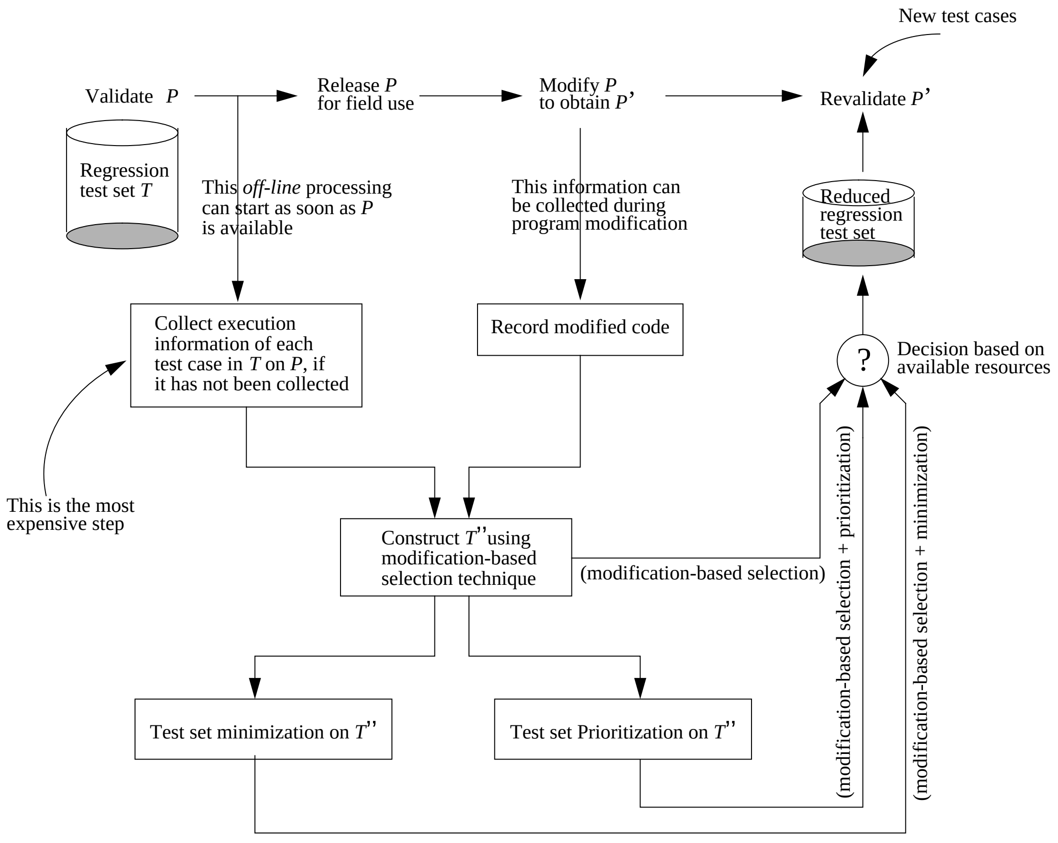
New test cases
Validate P
Release P for field use
Modify P, to obtain P'
Revalidate P'
Regression test set T
This off-line processing can start as soon as P is available
This information can be collected during program modification
Reduced regression test set
Collect execution information of each test case in T on P, if it has not been collected
Record modified code
?
Decision based on available resources
This is the most expensive step
Construct T'' using modification-based selection technique
(modification-based selection)
(modification-based selection + prioritization)
(modification-based selection + minimization)
Test set minimization on T''
Test set Prioritization on T''

Figure 1. Selecting effective regression test cases through off-line processing

too expensive for regression testing. What additional procedures can be applied to further reduce the number of tests to be used ? Randomly discarding tests is a possible solution. However, since this solution may arbitrarily remove the tests that execute certain modifications in code, there is an arbitrarily high risk of eliminating all tests that reveal an error in the code.

To solve this problem, two techniques, *minimization* and *prioritization*, are proposed. A test set minimization procedure finds a minimal subset in terms of the number of test cases that preserves the coverage with respect to a certain criterion of the original test set. Only test cases in the minimized subset are used for software revalidation. A test set prioritization procedure sorts test cases in order of increasing cost per additional coverage, and then selects the top n test cases for revalidation. The rationale behind these two techniques is the following:

1. The correlation between fault detection effectiveness and code coverage is higher than that between fault detection effectiveness and test set size [22]. Insufficient testing such as long hours of test case execution that do not increase code coverage can lead to incorrect conclusions on software validation.

2. No matter how a test set is generated (with or without any test case screening), its minimized test sets have a size/effectiveness advantage over the original set in terms of fewer test cases with little or no compromise in the strength of revealing faults [23, 24].

3. In general, a reduced set of tests selected by minimization may detect faults not detected by a reduced set in the same size selected in a random or arbitrary way. Null hypothesis testing reported in [23, 24] indicates that such effectiveness advantage of minimization over randomization does not just happen by chance.

One problem of the prioritization approach is that we do not know how much is enough while selecting test cases one by one from the top. Unlike the minimization approach, which uses every test in the minimized set, the *magic* number of tests in a prioritized set is unknown. Thus, time and budget permitting, testers may, on the basis of test case priority, run as many tests as possible.

The *possible* advantage of prioritization is that its time complexity, $O(n^2)$ for the worst case, is better than that for test set size minimization, which is equivalent to the NP-complete “minimal set covering problem” [7] and may re-

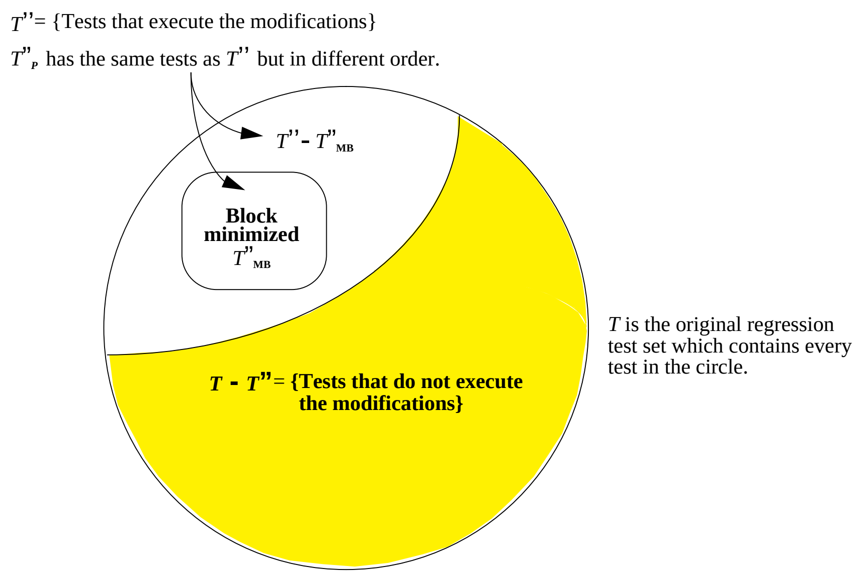
T''= {Tests that execute the modifications}
T''_P has the same tests as T'' but in different order.
T'' - T''_MB
Block minimized T''_MB
T is the original regression test set which contains every test in the circle.
T - T''= {Tests that do not execute the modifications}

Figure 2. Subsumption among the original regression test set (T), modification-based set (T''), and the corresponding block minimized set (T''_{MB}). Regression tests in the shaded area (i.e., T - T'') do not need to be reexecuted.

quire an exponential time on some "set covering problems" derived from Steiner triples [6].

2.3 ATAC: an Automatic Testing Analysis tool in C

A tool called ATAC [12] was used to conduct the modification-based test selection and to find the subsequent minimized or prioritized sets. Given a program to test, ATAC computes its set of testable attributes such as block, decision, p-use, c-use and all-uses [11, 16], and instruments it to record trace information during test execution. As new tests are executed, such information is appended to a trace file. ATAC then uses this trace file and the corresponding *atac* files which provide locations of each test attribute to select test cases according to the program modification.

For test set prioritization, ATAC first selects a test case which gives the maximal coverage with respect to a given criterion per unit cost. Subsequent tests are selected based on their *additional* coverage per unit cost.

For test set minimization, ATAC uses an implicit enumeration algorithm with reductions to find the optimal subset based on all tests examined. Although, theoretically, this algorithm can be exponential, no problem derived from test set minimization has been any more costly to solve than by inexact greedy heuristics [4, 9, 25].[2] Examples of this include using ATAC to conduct test set minimization on (1) an object-oriented language compiler (100 KLOC), (2) a provisioning application (353 KLOC) with 32K regression tests, and (3) a space application (10 KLOC) developed for the European Space Agency [24]. In addition, researchers and testers at another corporation have also applied ATAC minimization to a widely used large application with 50 files (35KLOC) and have not encountered any excessive computing time.

[2] Besides determining the exact minimized set, ATAC also provides options to select an approximate minimized set based on "greedy" heuristics in the event that the exact solution is not obtained in reasonable time.

2.4 Sequence of test case selection

In summary, to conduct efficient and effective regression testing, we recommend the following sequence for selecting tests from the existing regression suite.

1. First, construct a superset of all regression tests that should be used to ensure that a new program preserves the desired functionality of the old program. Such construction is done by a modification-based test selection.

2. Second, if necessary, use minimization or prioritization for further test screening based on those selected by modification. Minimization, for example, can be done with respect to the block coverage. For a given test set, block minimization returns a minimal subset in terms of the number of test cases such that the block coverage is preserved. The Venn diagram in Figure 2 shows the subsumption among the original regression test set, modification-based set, and the corresponding block minimized set. In the case of prioritiza-

tion, according to the time pressure and budget constraint, rerun as many tests as possible starting from the top of the prioritized list. Hereafter, T''_{MB} and T''_P, respectively, are used as the block minimized subset and the prioritized set with respect to a given T'', a modification-based selected set.

The above two steps provide a general guideline to decide which regression tests have a higher priority to be reused, and which can be omitted or have a lower priority. However, in a situation such as when certain tests are known to exercise particularly important features or address earlier bugs, it might be a good idea to use them also to test the modified program. If these tests are in the superset[3] generated at the first step, ATAC can guarantee the inclusion of such important tests in the minimized set or list them at the top in the prioritized set so that they will be reused. Such a guarantee is accomplished by assigning a zero cost to these tests.

3 A Case Study

A case study was conducted on a space program developed for the European Space Agency.[4] A modification-based selection technique and the subsequent test set minimization and prioritization techniques were used to select regression tests for reexecution. Three metrics, *size reduction*, *recall* and *precision*, were computed to measure the cost-effectiveness of these techniques. Our experiment is explained below in detail.

3.1 The Space Program

The space program provides a language-oriented user interface that allows the user to describe the configuration of an array of antennas using a high level language [2]. Its purpose is to prepare a data file in accordance with a predefined format and characteristics from a user, given the array antenna configuration described in a language-like form. An appropriate Array Definition Language was defined and used within the program. This language allows the user to describe a certain antenna array by a few statements instead of having to write the complete list of elements, positions and excitations. The program consists of about 10,000 lines of code divided into three subsystems, parser, computation, and formatting. Details of these subsystems can be found in their design documents [2].

[3] These tests will not be selected by the modification-based technique if they do not execute any code that changes, with respect to the original program, for the modified program. Actually, such tests will not be selected by any technique which selects regression tests based on the output difference between the original program and the modified program.

[4] We thank Dr. Alberto Pasquini for providing us the source code, test cases and error-log files of the space program.

The fault set for the space program used in this study was obtained from the error-log maintained during its testing and integration phase (see Appendix). For convenience, each fault has been numbered as **Fk** where the integer **k** denotes the fault number. This number does not indicate the order in which faults were detected. Table 1 lists a brief classification, based on a scheme in reference [19], of each fault.

Table 1. Fault classification

Fault	Classification	
	Type	Subtype
F1	Logic omitted or incorrect	Missing condition test
F2	Logic omitted or incorrect	Missing condition test
F3	Computational problems	Equation insufficient or incorrect
F4	Computational problems	Equation insufficient or incorrect
F5	Data handling problems	Data accessed or stored incorrectly
F6	Computational problems	Equation insufficient or incorrect
F7	Computational problems	Equation insufficient or incorrect
F8	Logic omitted or incorrect	Missing condition test
F9	Logic omitted or incorrect	Checking wrong variable
F10	Logic omitted or incorrect	Checking wrong variable

3.2 Regression Test Suite

A regression test suite of 1000 distinct tests was created based on the operational profile of how the space program was used. An operational profile, as formalized by Musa [14] and used in our experiment, is a set of the occurrence probabilities of various software functions. To obtain an operational profile for `space` we identified the possible functions of the program and generated a graph capturing the connectivity of these functions. Each node in the graph represented a function. Two nodes, A and B, were connected if control could flow from function A to function B. There was a unique start and end node representing functions at which execution began and terminated, respectively. A path through the graph from the start node to the end node represents one possible program execution. To estimate the occurrence probability of the software functions, each arc was assigned a transition probability, i.e., the probability of control flowing between the nodes connected by the arc. For example, if node A is connected to nodes B, C and D, and the probabilities associated with arcs A-B, A-C, and A-D are, respectively, 0.3, 0.6 and 0.1, then after the execution of function A the program will perform functions B, C or D, respectively, with probability 0.3, 0.6, and 0.1. There was a total of 236 function nodes. Transition probabilities were determined by interviewing the program users.

3.3 Results

Regression tests were selected according to the modification for each fault listed in the Appendix. Block minimization with respect to the entire space program was performed on these selected tests. Test set prioritization was also conducted to prioritize such regression tests based on their additional block, decision and all-uses coverage per unit cost. For the purpose of illustration, regression tests in the the block minimized and prioritized sets with respect to those selected using the modification-based technique on fault F1 are listed in Table 2.[5]

3.4 Analysis

Data collected from our experiment were analyzed to determine whether the small set of regression tests selected for reexecution according to the sequence suggested in Section 2.4 provides a large *size reduction* as well as good *precision* and *recall*.

3.4.1 Size reduction

For each fault, Table 3 lists the number of regression tests in T' on which the old and new programs produce different outputs and the number of tests in T'' selected according to the modification-based technique. The size of subsequent prioritized (T''_P) and block minimized (T''_{MB})[6] sets is also included. Sets T'' and T''_P have the same regression tests but in different order. Two reductions in size, with respect to T' and T'', respectively, were computed. Table 3 lists the reductions for T''_{MB} and Table 4 lists the reductions for T''_P with three different $\mathcal{N}$ values. These values are defined as one-third of, two-thirds of, and the same as the size of T''_{MB}, wherever possible. Exceptions occur in some situations. One example is in fault F1 where $\mid T''_{MB} \mid = 14$ and $\mathcal{N}_1$ equals 5 and $\mathcal{N}_2$ equals 10. Two other examples are in faults F4 and F5 where $\mid T''_{MB} \mid$ equals 2, and $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_3$ are 1, 2, and 3, respectively.

From the experimental data and the summary in Tables 3 and 4, it is observed that size reductions, in general, are greater than 59.09% with respect to T'' and 50.00% with respect to T' apart from a few exceptions. In fact, many of them are more than 70%.

The size reduction for F8 is a negative value when T' is considered. This is because only six of the 34 regression tests in T that execute the code in P, at which new code is added to prepare P', actually distinguish P' from P. Since the number of regression tests in T'' is almost six times larger than that in T', it is not unusual to have more tests in T''_{MB} than T'. The same reasoning applies to the negative size reduction in Table 4.

Except for the rare exception as discussed above, our data suggest T''_{MB} has a significant *size* advantage over T' and, of course, T'', too. This is also true for prioritized sets (T''_P) when only a few top tests are used for program revalidation.

3.4.2 Precision and recall

The size reduction computed in the previous section shows the savings of our regression test selection approach in terms of the number of tests. The possible loss accompanied by such selection is measured in terms of precision and recall defined as below.

> Suppose a regression test suite T contains r tests. Among these, n tests $(n \leq r)$ result in different output behavior for the old program P than for the new program P'. A test set $T' \subseteq T$ contains m $(m \neq 0)$ tests determined by using a regression test selection technique $\mathcal{C}$. Of these m tests, l tests can distinguish P' from P. The precision of T' relative to P, P', T and $\mathcal{C}$ is the percentage given by the expression $100{*}(l/m)$, whereas the corresponding recall is (i) the percentage given by the expression $100{*}(l/n)$ if $n \neq 0$ or (ii) 100% if $n = 0$.

Based on the above definition, the precision of a test set is the percentage of its tests on which the new and the old programs produce different outputs to the total number of tests in it. For example, T''_{MB} contains nine tests, seven of which can detect F2, that is, seven of which can make P' differ from P because of the bug-fixing on F2. The precision of T''_{MB} on F2 is computed as 100*(7/9)=77.78%. Similarly, the recall of a test set is the percentage of regression tests selected from those that need to be reexecuted. Let us use fault F2 as the example again. There are 16 tests in the regression suite that can detect F2, seven of which are in T''_{MB}. This makes the recall of T''_{MB} on F2 equal to 100*(7/16)=43.75%.

Precisions and recalls for T''_{MB} and the top $\mathcal{N}$ tests from T''_P relative to each fault are listed in Tables 5 and 4, respectively. From these data, we observed:

1. The precisions in general are greater than 60%, apart from a few exceptions. More than half of them are above 70%. Those for faults F4 and F5 are 100% in every case.

2. There is a large variation in recall when different faults are considered. It can be as high as 100%, such as the one for T''_{MB} on F8; it can also be less than 4%, such as that for F10. The explanation for this is that only six regression tests can detect F8, whereas 320 tests can detect F10.

[5] A complete data set may be obtained by sending electronic mail to W. Eric Wong at ewong@bellcore.com.

[6] Data for minimization with respect to (1) block and decision coverage, and (2) block, decision, and all-uses coverage are also computed but not included due to the space limitations.

Table 2. Regression tests in the block minimized and prioritized sets with respect to the modification based selection for fault F1

T''_{MB}	T''_{P}			
Selected regression tests	Cumulative coverage			Selected regression tests
	Block (%)	Decision (%)	All-uses (%)	
case-555	46.04	30.02	33.35	case-555
case-620	53.78	36.26	39.88	case-48
case-407	60.18	43.42	46.23	case-207
case-995	61.48	45.62	48.24	case-733
case-31	63.03	47.39	49.80	case-120
case-120	63.77	48.65	51.24	case-620
case-733	64.61	49.66	52.04	case-618
case-618	64.90	50.08	52.41	case-143
case-756	65.04	50.42	52.78	case-945
case-941	65.18	50.67	53.17	case-995
case-804	65.28	50.93	53.42	case-804
case-945	65.39	51.26	53.62	case-756
case-156	65.39	51.35	53.82	case-821
case-754	65.42	51.43	53.99	case-31
	65.49	51.60	54.10	case-156
	65.53	51.69	54.20	case-754
	65.53	51.69	54.27	case-284
	65.53	51.77	54.29	case-677
	65.53	51.77	54.31	case-763[*]
	65.53	51.77	54.31	case-19
	65.53	51.77	54.31	case-441
	65.53	51.77	54.31	case-407
	65.53	51.77	54.31	case-351
	65.53	51.77	54.31	case-303
	65.53	51.77	54.31	case-289
	65.53	51.77	54.31	case-864
	65.53	51.77	54.31	case-247
	65.53	51.77	54.31	case-941
	65.53	51.77	54.31	case-200
	65.53	51.77	54.31	case-539
	65.53	51.77	54.31	case-930
	65.53	51.77	54.31	case-879
	65.53	51.77	54.31	case-117
	65.53	51.77	54.31	case-93
	65.53	51.77	54.31	case-71
	65.53	51.77	54.31	case-69
	65.53	51.77	54.31	case-877
	65.53	51.77	54.31	case-710

[*] Those after case-763 in the prioritized set do not add any cumulative coverage with respect to block, decision and all-uses.

Table 3. Size comparison among various regression sets

Fault	Number of regression tests				Size reduction with respect to T'	Size reduction with respect to T''
	$\mid T' \mid$	$\mid T'' \mid$	$\mid T''_P \mid$	$\mid T''_{MB} \mid$	$(\mathcal{R}'_{MB})$	$(\Re''_{MB})$
F-1	26	38	38	14	46.15	63.16
F-2	16	22	22	9	43.75	59.09
F-3	36	84	84	12	66.67	85.71
F-4	4	4	4	2	50.00	50.00
F-5	4	4	4	2	50.00	50.00
F-6	32	38	38	9	71.88	76.32
F-7	32	38	38	9	71.88	76.32
F-8	6	34	34	9	-50.00	73.53
F-9	46	71	71	15	67.39	78.87
F-10	320	470	470	18	94.38	96.17

• $\mathcal{R}'_{MB} = (1 - \frac{|T''_{MB}|}{|T'|}) * 100$ and $\Re''_{MB} = (1 - \frac{|T''_{MB}|}{|T''|}) * 100$

Table 4. Characteristics of the top $\mathcal{N}$ tests in T''_P

Fault	$\mathcal{N}_1$	Size reduction		Precision	Recall
		$\mathcal{R}'_{\mathcal{N}_1}$	$\Re''_{\mathcal{N}_1}$	(%)	(%)
F-1	5	80.77	86.84	80.00	15.38
F-2	3	81.25	86.36	66.67	12.50
F-3	4	88.89	95.24	50.00	5.56
F-4	1	75.00	75.00	100.00	25.00
F-5	1	75.00	75.00	100.00	25.00
F-6	3	90.62	92.11	66.67	6.25
F-7	3	90.62	92.11	66.67	6.25
F-8	3	50.00	91.18	33.33	16.67
F-9	5	89.13	92.96	60.00	6.52
F-10	6	98.12	98.72	66.67	1.25

Fault	$\mathcal{N}_2$	Size reduction		Precision	Recall
		$\mathcal{R}'_{\mathcal{N}_2}$	$\Re''_{\mathcal{N}_2}$	(%)	(%)
F-1	10	61.54	73.68	70.00	26.92
F-2	6	62.50	72.73	83.33	31.25
F-3	8	77.78	90.48	37.50	8.33
F-4	2	50.00	50.00	100.00	50.00
F-5	2	50.00	50.00	100.00	50.00
F-6	6	81.25	84.21	83.33	15.62
F-7	6	81.25	84.21	83.33	15.62
F-8	6	0.00	82.35	16.67	16.67
F-9	10	78.26	85.92	40.00	8.70
F-10	12	96.25	97.45	66.67	2.50

Fault	$\mathcal{N}_3$	Size reduction		Precision	Recall
		$\mathcal{R}'_{\mathcal{N}_3}$	$\Re''_{\mathcal{N}_3}$	(%)	(%)
F-1	14	46.15	63.16	71.43	38.46
F-2	9	43.75	59.09	77.78	43.75
F-3	12	66.67	85.71	50.00	16.67
F-4	3	25.00	25.00	100.00	75.00
F-5	3	25.00	25.00	100.00	75.00
F-6	9	71.88	76.32	77.78	21.88
F-7	9	71.88	76.32	77.78	21.88
F-8	9	-50.00	73.53	22.22	33.33
F-9	15	67.39	78.87	53.33	17.39
F-10	18	94.38	96.17	72.22	4.06

- $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ are, respectively, one-third of, two-thirds of, and the same as $\mid T''_{MB} \mid$, wherever possible.
- $\mathcal{R}'_{\mathcal{N}_i}$ computed as $(1 - \frac{\mathcal{N}_i}{|T'|}) * 100$, for $i = 1, 2, 3$ is the size reduction with respect to T'.
- $\Re''_{\mathcal{N}_i}$ computed as $(1 - \frac{\mathcal{N}_i}{|T''|}) * 100$, for $i = 1, 2, 3$ is the size reduction with respect to T''.

Table 5. Characteristics of block minimized sets (T''_{MB})

Fault	Precision (%)	Recall (%)
F-1	64.29	34.62
F-2	77.78	43.75
F-3	50.00	16.67
F-4	100.00	50.00
F-5	100.00	50.00
F-6	77.78	21.88
F-7	77.78	21.88
F-8	66.67	100.00
F-9	53.33	17.39
F-10	50.00	2.81

In general, if there are many regression tests that need to be reexecuted, most of them are not expected to be included in T''_{MB} or the top few tests in T''_P. Otherwise, the merit of a trade-off among size reduction, precision and recall is lost. Under such a scenario, a low percentage of recall is certainly expected. For a given test set, the absolute magnitude of its precision and recall with respect to certain faults (or more precisely the resulting modifications on fixing such bugs) should not be interpreted without taking its size reduction into account also. A small percentage in recall is not necessarily inappropriate; on the contrary, it may indicate a significant savings in terms of the number of tests, if the corresponding precision is reasonably high.

4 Discussion

Effective regression testing is a trade-off between the number of regression tests needed and the cost. The greater the number of regression tests, the more complete the program revalidation. However, this also requires a larger budget and greater resources——which may not be affordable in practice. Running fewer regression tests may be less expensive, but has the potential of not being able to ensure that all the inherited features still behave as expected, except where changes are anticipated. In this paper, we propose a modification-based technique followed by test set minimization or prioritization to determine which regression tests should be rerun. Stated differently, only regression tests in (I) T''_{MB} obtained by modification + minimization, or (II) the top $\mathcal{N}$ from T''_P obtained by modification + prioritization are selected.

Three metrics, *size reduction*, *precision*, and *recall*, are used to examine the goodness of using (I) and (II). These metric values depend not only on the nature of the regression test suite but also the extent and the locations of the modifications. For example, if a modification, such as for `F10`, affects the program output for many test cases, then a big size reduction and a small recall are expected. Another example is that if a modification made in a program, such as for `F4` and `F5`, is rarely executed, but each time when it is executed it makes a change in the program output, then a high precision is expected. The overall results from our case study suggest both (I) and (II) can serve as good, cost-effective alternatives for testers who need to conduct quick regression testing under time pressure and budget constraints.

Unlike many other *proposed* test selection techniques, ours is supported by a tool called ATAC. As explained in Section 2.3, we have not experienced any excessive computing time in doing test set minimization. If the exact solution is not obtained in reasonable time, ATAC provides an approximate minimized set using "greedy" heuristics. In fact, our data indicate that in many cases the top $\mathcal{N}$ tests from T''_P, with $\mathcal{N} = \mid T''_{MB} \mid$, have the same precision and recall as T''_{MB}. The trouble is that without computing T''_{MB}, we do not know the magic number $\mathcal{N}$. Another way to combine test set minimization and prioritization is to run every test case in T''_{MB} first, followed by some top tests selected from the prioritized list with respect to those in T'' - T''_{MB}. The advantage of doing so is that we always select regression tests in the order of increasing cost per additional coverage.

If for reasons of safety or performance an application is not tolerant of a great deal of intrusive instrumentation, our modification-based test selection can be conducted at a higher level of granularity such as at the function or subsystem level. The disadvantage is that the resulting T''_{MB} or the top $\mathcal{N}$ tests from T''_P may suffer a lower precision and a lower recall because more regression tests that do not distinguish P' from P may be included in T''. On the other hand, if a high percentage of precision and recall is required, one can apply more complicated, and more expensive, techniques such as relevant slicing [1] to construct a *smaller* super set, $\mho$, of those regression tests which need to be reexecuted. We can then apply test set minimization and prioritization to $\mho$ to identify a representative subset of tests which have a higher priority to be rerun.

The bottom line for making a decision on whether any of these alternatives should be adopted goes back to the original *trade-off* problem: What we should do in regression testing versus what we can afford to do. Experiments are underway to compare the cost-effectiveness of these alternatives. The results of these studies will provide more information to help us determine how to conduct efficient and effective regression testing in practice.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of the Conference on Software Maintenance*, pp 348-357, Montréal Quebec, Canada, September 1993.

[2] A. Cancellieri and A. Giorgi, "Array preprocessor user manual," Technical Report IDS-RT94/052, 1994.

[3] Y. F. Chen, D. S. Rosenblum, and K. P. Vo, "TestTube: A system for selective regression testing," in *Proceedings of the 16th IEEE International Conference on Software Engineering*, pp 211-222, Sorrento, Italy, May 1994.

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

[5] K.F. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *Proceedings of the National Telecommunications Conference*, pp B6.3.1-B6.3.6. IEEE Computer Society Press, 1981.

[6] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter Jr, "Two computational difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems," *Mathematical Programming Study*, 2:72-81, 1974.

[7] M. R. Gary and D. S. Johnson, "*Computers and Intractability*," Freeman, New York, 1979.

[8] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proceedings of the Conference on Software Maintenance*, pp 299-308, Orlando, FL, November 1992.

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

[10] J. Hartmann and D. J. Robson, "Techniques for selective revalidation," *IEEE Software*, pp 31-36, January 1990.

[11] J. R. Horgan and S. A. London, "Data flow coverage and the C language," in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pp 87-97, Victoria, British Columbia, Canada, October 1991.

[12] J. R. Horgan and S. A. London, "ATAC: A data flow coverage testing tool for C," in *Proceedings of Symposium on Assessment of Quality Software Development Tools*, pp 2-10, New Orleans, LA, May 1992.

[13] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *Proceedings of the Conference on Software Maintenance*, pp 282-290, Orlando, FL, November 1992.

[14] J. D. Musa, “Operational profiles in software reliability engineering,” *IEEE Software*, 10(2):14-32, March 1993.

[15] 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*, pp 233-247, September 1988.

[16] S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information,” *IEEE Trans. on Software Engineering*, SE-11(4):367-375, April 1985.

[17] G. Rothermel, “*Efficient, Effective Regression Testing Using Safe Test Selection Techniques*,” PhD thesis, Department of Computer Science, Clemson University, Clemson, SC, May 1996.

[18] B. Sherlund and B. Korel, “Modification oriented regression testing,” in *Proceedings of the Fourth International Software Quality Week*, pp 149-158, San Francisco, CA, May 1991.

[19] IEEE Computer Society, “IEEE 1044 - standard classification for software errors, faults and failures,” *IEEE Computer Society*, 1994.

[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*, pp 527-534. IEEE Computer Society Press, 1989.

[21] L. J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha, “Test manager: A regression testing tool,” in *Proceedings of the Conference on Software Maintenance*, pp 338-347, Montreal Quebec, Canada, September 1993.

[22] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, “Effect of test set size and block coverage on fault detection effectiveness,” in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pp 230-238, Monterey, CA, November 1994.

[23] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, “Effect of test set minimization on fault detection effectiveness,” in *Proceedings of the 17th IEEE International Conference on Software Engineering*, pp 41-50, Seattle, WA, April 1995.

[24] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, “Test set size minimization and fault detection effectiveness: A case study in a space application,” in *Proceedings of the 21st Annual International Computer Software and Application Conference*, pp 522-528, Washington, D.C., August 1997.

[25] Y. A. Zuev, “A set-covering problem: The combinatorial-local approach and the branch and bound method,” *U.S.S.R Computational Mathematics and Mathematical Physics*, 19(6):217-226, June 1979.

Appendix: Faults considered in this study

Fault	File	Line	Incorrect string	Correct string
F1	sgramp2n.c	122	a = ((p1_et + p2_et - 2 * e)/(2 * x1 * x1));	if (x1 == 0)
			c = ((a * x1 * x1 + e - p1_et)/x1);	{a = c = 0;}
			b = ((q1_et + q2_et - 2 * e)/(2 * y1 * y1));	else {a = ((p1_et + p2_et - 2 * e) / (2 * x1 * x1));
			d = ((b * y1 * y1 + e - q1_et)/y1);	c = ((a * x1 * x1 + e - p1_et) / x1);
				};
				if (y1 == 0)
				{b = d = 0;}
				else {b = ((q1_et + q2_et - 2 * e) / (2 * y1 * y1));
				d = ((b * y1 * y1 + e - q1_et) / y1);
				};
F2	sgrpha2n.c	123	a = ((p1_ep + p2_ep - 2 * e) / (2 * x1 * x1));	if (x1 == 0)
			c = ((a * x1 * x1 + e - p1_ep) / x1) ;	{a = c = 0;}
			b = ((q1_ep + q2_ep - 2 * e) / (2 * y1 * y1));	else {a = ((p1_ep + p2_ep - 2 * e) / (2 * x1 * x1));
			d = ((b * y1 * y1 + e - q1_ep) / y1) ;	c = ((a * x1 * x1 + e - p1_ep) / x1);
				};
				if (y1 == 0)
				{b = d = 0;}
				else {b = ((q1_ep + q2_ep - 2 * e) / (2 * y1 * y1));
				d = ((b * y1 * y1 + e - q1_ep) / y1);
				};
F3	mkshex.c	84	x = P[i] * pstep + Q[i] * qstep * cos(angle);	x = P[i] * pstep + Q[i] * qstep * dcos(angle);
			y = Q[i] * qstep * sin(angle);	y = Q[i] * qstep * dsin(angle);
F4	sgrrot.c	42	XE = ((XD - XC) * cos(phi)) -	XE = ((XD - XC) * dcos(phi)) -
			((YD - YC) * sin(phi)) + XC ;	((YD - YC) * dsin(phi)) + XC ;
			YE = ((XD - XC) * sin(phi)) +	YE = ((XD - XC) * dsin(phi)) +
			((YD - YC) * cos(phi)) + YC ;	((YD - YC) * dcos(phi)) + YC ;
F5	sgrrot.c	54	app_ptr->PSEA += phi;	app_ptr->PHEA += phi;
F6	seqrothg.c	49	can = angle_step;	can += angle_step;
F7	seqrothg.c	42	cph = phase_step;	cph += phase_step;
F8	seqrotrg.c	52	Missing code	if ((pmin == pmax) && (qmin == qmax)) {
				gnodevis(pmin, qmin, cph, can, g);
				cph += phase_step;
				can += angle_step;
				cont++;
				if (cont == nodes_num)
				endvisit = 1;
				}
				else {
		121	Missing code	};
F9	pqlimits.c	28	while (app_ptr->NEXT != NULL) {	while (app_ptr != NULL) {
F10	mksblock.c	68	for (q = q1; q <= q2; q++) {	for (q = intmin(q1,q2); q <=intmax(q1,q2); q++) {
			for (p = p1; p <= p2; p++) {	for (p = intmin(p1,p2); p <=intmax(p1,p2); p++) {