

Testing Evolving Software*

Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
harrold@cis.ohio-state.edu

Abstract

Regression testing, which attempts to validate modified software and ensure that no new errors are introduced into previously tested code, is used extensively during maintenance of evolving software. Despite efforts to reduce its cost, regression testing remains one of the most expensive activities performed during a software system's lifetime. Because regression testing is important and expensive, many researchers have focused on ways to make it more efficient and effective. Research on regression testing spans a wide variety of topics, including test environments and automation, capture-playback mechanisms, regression-test selection, coverage identification, test suite maintenance, regression testability, and regression-testing process. This paper discusses the state of the art in several important aspects of regression testing, and presents some promising areas for future research.

1 Introduction

Software maintenance can account for as much as two-thirds of the cost of software production [3, 15]. This expense occurs in part because, in today's market, software development and testing are dominated by rework of existing software, not design of new software [3]. *Regression testing*, which attempts to validate modified software and ensure that no new errors are introduced into previously tested code, is used extensively during this evolution process. Regression testing is used to test safety-critical software that must be retested often, to test software that is being developed under constant evolution as the market or technology changes, to test new or modified components of a system, and to test new members in a family of similar products. Despite efforts to reduce its cost, regression testing remains one of the most expensive activities performed during a software system's lifetime.

Because regression testing is expensive, but important, researchers have focused on ways to make it more efficient and effective. Research on regression testing spans a wide variety of topics. Test environments and automation (e.g., [13]), and capture-playback mechanisms (e.g., [17]) provide support for regression testing. Techniques for regression-test selection (e.g., [1, 4, 12, 30]), coverage identification (e.g., [12, 19, 28]), and test suite maintenance (e.g., [9, 27, 34, 35]) facilitate selective testing of the modified software. Regression testability permits estimation, prior to regression test selection, of the number of tests that will be selected by a method (e.g., [10, 15, 24]), or evaluation, prior to implementation, of the difficulty of regression testing (e.g., [8, 31]). A regression-testing process (e.g., [18]) can integrate many of these key techniques into development and maintenance of the evolving software.

Some of these techniques are already being used in practice. For example, many companies have used capture-playback techniques to automate part of their regression-testing process. Most of this technology,

*This work was supported in part by grants from Microsoft Inc. and Boeing Airplane Group, and by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University.

however, is not being used in practice, in part because the scalability and the usefulness of the techniques have not been convincingly demonstrated. The increased awareness by researchers of the importance of empirical evaluation and the support for such empirical work by agencies, such as the National Science Foundation,¹ have yielded initial empirical data on the cost benefits of these techniques. In addition to showing the potential scalability and usefulness of some of the techniques, these data also highlight the need for additional evaluation of existing techniques and for development of modified and new techniques.

The increased interest in techniques that help to reduce the costs of regression testing, the maturity of the research in regression testing, and the commitment to empirical evaluation of regression-testing techniques should hasten the transfer of regression-testing technology to industry. This paper discusses the state of the art, existing empirical results, and promising future work in selective retest, one aspect of regression testing for which I believe industrial-strength tools will soon be available.

2 Selective Retest in a Maintenance Process

During regression testing, a test suite T , used to test a program P^2 , and information about the results of testing P with T are available. *Selective-retest* techniques attempt to reduce the cost of regression testing by reusing T and identifying portions of the modified program P' or its specification that should be tested. For example, a selective-retest technique may select all tests from T that execute code that is new or modified from P to P' . Selective retest techniques differ from a *retest-all* approach, which runs all tests in T or reanalyzes and retests all of P' . Leung and White [16] show that a selective-retest technique is more economical than a retest-all technique only if the cost of selecting a reduced subset T' of tests from T is less than the cost of running the tests in $T - T'$.

Selective-retest can be integrated into a maintenance process that consists of the following steps:³

1. Identify parts of P that will be changed to produce P' .
2. Modify P to get P' , which may involve modifying other documents, such as the requirements document.
3. Select $T' \subseteq T$ to execute on P' .
4. Test P' with T' to
 - (a) establish P' 's correctness with respect to T' or
 - (b) identify tests in T' that fail for P' , identify the faults that caused the failures, and restart the process at step 1.
5. If necessary, create T'' , a set of new functional or structural tests developed to test new, modified, or untested parts of P' .
6. Test P' with T''
 - (a) establish P' 's correctness with respect to T'' or
 - (b) identify tests in T'' that fail for P' , identify the faults that caused the failures, and restart the process at step 1.
7. Create $T''' = T \cup T''$, a new test suite for P' .

¹Evidence of the increased awareness of the importance of experimental work is seen by the support of the National Science Foundation for significant grants for experimental research and for the Workshop on Empirical Research in Software Engineering, which was held during Summer 1998.

² P could be either a procedure or a program.

³Both Rothermel and Harrold [30] and Onoma et al. [18] present a discussion of the steps involved in selective retest. In this paper, we integrate these steps into one maintenance process.

In performing these steps, this maintenance process addresses several problems. Step 1 involves the *modification-request problem*: the problem of gaining approval for a change. This step may involve configuration management and impact analysis. Step 2 involves the *software-update problem*: the problem of modifying the appropriate documents and code to reflect the change. This step may involve issues in program understanding. Steps 4(b) and 6(b) involve the *fault-identification problem*: the problem of debugging the software or tests to identify the faults. Step 3 involves the *regression-test-selection problem*: the problem of selecting a subset T' of T with which to test P' . This problem includes the subproblem of identifying tests in T that are obsolete for P' .⁴ Step 5 addresses the *coverage-identification problem*: problem of identifying portions of P' or S' that require additional testing. Steps 4 and 6 address the *test-execution problem*: the problem of efficiently executing tests and checking test results for correctness. Step 7 addresses the *test-suite maintenance problem*: the problem of updating and storing test information; it also addresses the *test-suite minimization problem*: the problem of minimizing the test suite by removing obsolete and redundant tests.

3 Selective-Retest Techniques

Although each of the problems discussed in the preceding section is significant for maintenance, this paper focuses on three important selective-retest problems: the regression-test-selection problem, the coverage-identification problem, and the test-suite-minimization problem.

3.1 Regression Test Selection

Regression-test-selection techniques attempt to reduce the cost of regression testing by selecting T' and using T' to test P' . Testing professionals are reluctant, however, to omit any tests from T that might cause P' to expose faults. *Safe* regression-test-selection techniques ensure that the test suite selected, T' , contains all tests in T that execute code that was modified from P to P' . We call these tests *modification-traversing* for P and P' . Rothermel [25] shows that the problem of precisely identifying modification-traversing tests in T is PSPACE-hard. Thus, unless $P=NP$, no efficient regression-test-selection algorithm will always identify precisely the tests that are modification-traversing for modifications from P to P' . Several safe regression-test-selection techniques that vary in precision and efficiency have been presented: Rothermel and Harrold present techniques based on control flow; Ball also presents techniques based on control flow; Vokolos and Frankl present a technique based on textual differencing; and Chen, Rosenblum, and Vo present a technique based on code entities.

A number of regression-test-selection techniques are *unsafe* in that they may omit tests from T' that are modification-traversing. Although unsafe, these techniques can often achieve test selection that is close to that achieved by safe techniques. Additionally, these techniques may provide information that may be useful for other selective-retest activities, such as coverage identification (see Section 3.2).

To illustrate the differences in the various techniques, we use program P , its modified version P' , and the control-flow graphs G and G' for P and P' , respectively, shown in Figure 1; the figure also shows the paths in G traversed by each of tests t_1, \dots, t_4 . From P to P' , C is modified to get C' , J is added, and H is duplicated as H_1 and H_2 .

⁴Test t is *obsolete* for P' if t specifies an input to P' that, according to P' 's specification, is invalid for P' , or t specifies an invalid input-output relation for P' .

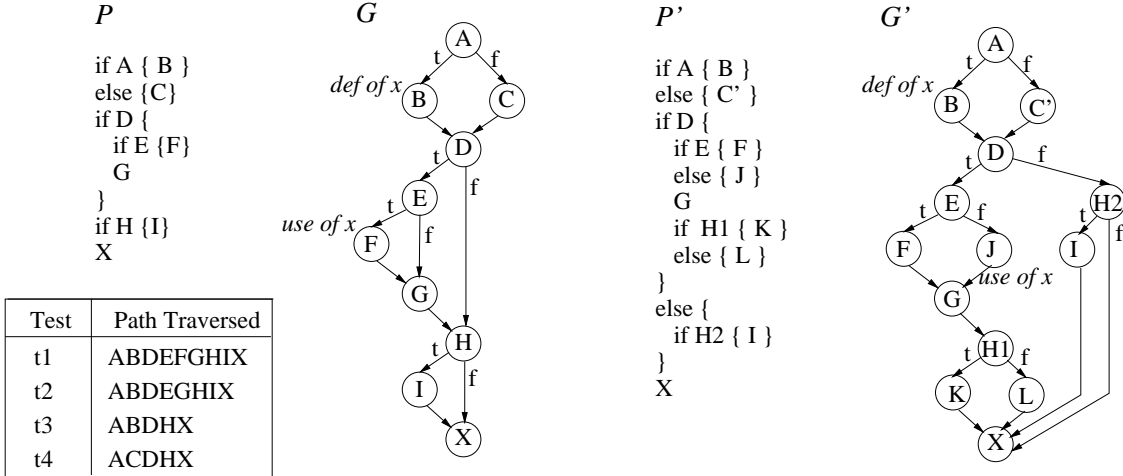


Figure 1: Example programs P and P' , their corresponding control-flow graphs G and G' , and information about the path traversed in G when P is executed with T .

In previous work, we present a regression-test-selection algorithm that uses G and G' , along with execution traces that associate tests in T with edges in G , to select tests for inclusion in T' [30]. We implemented our algorithm in a tool, **DejaVu**, that performs regression test selection for C programs. We proved that, under certain well-defined conditions, our test selection algorithm is safe. Under these conditions, the fault-detection abilities of T' are equivalent to the retest-all approach. We also showed that if, during the test selection, our algorithm does not multiply visit any pairs of vertices in G and G' , our algorithm is edge-optimal — it selects the least number of tests possible using a trace that associates tests from T with edges in G . The algorithm traverses corresponding paths in G and G' until, as it finds that the sinks of like-labeled edges differ; at that point it selects all tests that executed the edge in G . To illustrate, consider Figure 1. To select tests for P' , **DejaVu** traverses subpaths along ABDE in G and G' . At E, when it considers ‘f’ labeled edges (E,G) in G and (E,J) in G' , it finds that the sinks of these edges, G and J, respectively, differ, and it selects t_2 , the test in T that executes edge (E,G), for inclusion in T' .

Ball [1] presents an edge-optimal algorithm that, under certain conditions, such as the case where our algorithm multiply-visits pairs of vertices in G and G' , provides more precision than our algorithm; he also presents additional algorithms based on control flow that are even more precise than the edge-based algorithms, at greater computation cost. His algorithm creates an intersection graph and uses it to select tests for inclusion in T' that will execute changes reachable from H1, and to omit tests from T' that will execute code reachable from H2.⁵ For example, when Ball’s algorithm is applied to G and G' of Figure 1, it selects t_1 and t_2 for the change reachable from H1. For the same change, **DejaVu** selects all four tests.

Vokolos and Frankl present a regression-test-selection algorithm that uses text differencing [32]. They implemented their algorithm in a tool called **Pythia**, that performs the regression test selection for C programs. Their algorithm maintains an association between basic blocks and tests in T , and compares the source files of P and P' to identify the modified program statements. Their algorithm performs the

⁵See [1] for details of the algorithm.

Table 1: Results of Test Selection for Four Safe, Regression-Test-Selection Algorithms

Change from P to P'	Tests Selected by TestTube	Tests Selected by Pythia	Tests Selected by DeJaVu	Tests Selected by Ball’s Algorithm
C is changed to C’	$t1, \dots, t4$	$t4$	$t4$	$t4$
J is added	$t1, \dots, t4$	$t1, t2$	$t2$	$t2$
H is duplicated as H1 and H2	$t1, \dots, t4$	$t1, \dots, t4$	$t1, \dots, t4$	$t1, t2$

comparison using UNIX `diff` utility. This algorithm is based on statements and not control flow; thus, it may select more tests than the control-flow-based algorithms, at a lesser computation cost. For example, on encountering new statement J, **Pythia** identifies the statement that precedes J, finds E, and selects $t1$ and $t2$, those tests that executed E in P . For the same change, **DeJaVu** and Ball’s algorithm select only $t2$.

Chen, Rosenblum, and Vo [4] present a regression-test-selection algorithm that detects modified code entities, which are defined as functions or as non-executable components, such as storage locations. They implemented the technique as a tool, called **TestTube**, that performs regression test selection for C programs. The technique selects all tests associated with changed entities. Because this technique is based on entities that are coarser-grained than those used by statement- or control-flow-based techniques, it may select more tests than those techniques, with lesser computation cost. For example, for all changes from P to P' in Figure 1, **TestTube** selects $t1, \dots, t4$, all tests in T .

Table 1 shows the results of test selection using all four approaches, and illustrates the relative selectivity of the approaches. Because **TestTube** selects at the function level, it selects all four tests for any change. For the modification from C to C’, all approaches, except **TestTube** are able to identify only $t4$ for inclusion in T' ; when there are no control-flow changes, **Pythia**, **DeJaVu**, and Ball’s algorithm select the same tests. For the addition of J, both **DeJaVu** and Ball’s algorithms are able to identify only $t2$ for inclusion in T' ; when there are control-flow changes, **Pythia** can produce less precise results than **DeJaVu** and Ball’s algorithm. For the duplication of H as H1 and H2, only Ball’s algorithm is able to select $t1$ and $t2$; in the case of multiply-visited nodes, **DeJaVu** can produce less precise results than Ball’s algorithm.

One subset of unsafe regression-test-selection techniques selects tests for inclusion in T' using associations between tests in T and test-coverage requirements based on the data flow in the program (e.g., [2, 12, 19]).⁶ For example, in previous work [12], we present a regression-test-selection technique that first associates tests in T with definition-use pairs⁷ in P , and then selects those tests from T that execute definition-use pairs that are associated with code that is modified or deleted from P to P' . To illustrate, consider P and G in Figure 1, and suppose that variable x is assigned a value in statement B (a *def* of x) and that this value of x is used in a computation in statement F (a *use* of x). Then (B, F) forms a test-coverage requirement based on the data flow in P , and $t1$, which executes this pair, is associated with that requirement. If test selection determines that this definition-use pair is affected by a change in P , then $t1$ is added to T' .

New code induces definition-use pairs in P' that are not present in P . Because these definition-use pairs

⁶See [29] for a thorough discussion of these techniques.

⁷A *definition-use pair* is a pair of statements ($S1, S2$), such that $S1$ defines some variable v , $S2$ uses v , and there is a path in the program from $S1$ to $S2$ along which v is not redefined.

do not exist for P , there are no tests in T associated with these new pairs, and thus, no tests in T are selected for inclusion in T' . For example, suppose that there is a use of x in the newly inserted statement J , and that (B, J) forms a new definition-use pair for x . Because this definition-use pair is new and there are no tests yet associated with it, the test-selection algorithm selects no tests in T to include in T' . Although t_2 will execute this definition-use pair in P' , it was not selected. Thus, this technique is unsafe.

To date, there have been a number of empirical studies that evaluate these regression-test-selection techniques. Using **DejaVu**, we investigated the costs and benefits of using our regression-test-selection algorithm [29, 26]. We used **DejaVu** to select tests for a variety of 100-500 line programs, for which savings averaged 45%, and for a larger (50,000 line) software system, for which savings averaged 95%. Our studies show that the cost effectiveness of test selection can vary widely based on a number of factors: the cost of analysis required to select the tests for T' , the cost of executing and validating T' on P' , the composition of T , and the nature of the modifications to P for P' . Our studies also show that, for the subjects used, there were no multiply-visited vertices. Thus, for these subjects, our algorithm is edge-optimal.

Rosenblum and Weyuker [24] used **TestTube** to select tests for 31 versions of the KornShell and its associated test suites. For 80% of the versions, **TestTube** selected 100% of the tests. The authors note, however, that the test suite they used contained only 16 tests, many of which caused all components of the system to be exercised.

Rosenblum and Rothermel [22] present the first comparative evaluation of two different regression-test-selection techniques, **DejaVu** and **TestTube**, on the same set of subjects. This study compared the relative precision of the two techniques; current work is underway to compare the relative efficiency of the two techniques. Their study suggests that, in some cases, the coarse-grained **TestTube** and the more fine-grained **DejaVu** produce similar reductions in the tests that can be selected. Their study also found, however, that **DejaVu** sometimes selects a test suite that is substantially smaller than the original test suite.

Vokolos and Frankl [33] performed an experiment to evaluate their regression-test-selection algorithm. In their experiment, they used 33 different versions of a C program of approximately 11,000 lines that had been used by the European Space Agency. The versions represented the correction of 33 different faults. They randomly created a test suite for use in the experiment. They performed the regression test selection using **Pythia**, and found that, for their subject program and versions, **Pythia** substantially reduced the size of the regression test suite. For example, in almost 50% of the versions, 80% reduction was achieved. They did not directly compare **Pythia** with **DejaVu** and they did not compare the precision of their results with those obtained using **DejaVu**.

Graves et al. [6] performed an experiment that compared a number of regression-test-selection techniques: minimization techniques (these attempt to select a minimal set of tests from T), safe techniques, data-flow-coverage-based techniques, random techniques, and retest all. They drew a number of observations from their experiment. First, minimization produced the smallest and least effective test suites. However, in cases where testing is very expensive, minimization may be cost-effective. Second, safe and data-flow methods

had nearly equivalent average behavior in terms of cost effectiveness, typically detecting the same faults, and selecting the same size test suite. Data-flow methods are more expensive if test selection is the only goal. However, they provide additional information that can be used for coverage identification, which may justify the additional cost in some cases. Third, they found that the test selection methods were not the only factors affecting their results; other factors include the programs, the nature of the modifications, and the composition of test suites.

In some cases, the regression-test-selection tools select a T' that contains almost all tests in T ; in these cases, the test selection may not be cost effective. Rosenblum and Weyuker [24] proposed *coverage-based predictors* for use in predicting the cost-effectiveness of selective regression-testing strategies. With such a predictor, a testing professional could quickly estimate the number of tests that would be selected by a test-selection algorithm. If the time to run the tests omitted is more than the estimated cost of analysis to select the tests, the tester can then run the test-selection algorithm to select the tests; if not, the tester can simply run all tests in T .

One of their predictors is used to predict whether a safe selective-regression-testing strategy will be cost-effective. Using the regression testing cost model of Leung and White [16], Rosenblum and Weyuker demonstrate the usefulness of this predictor by describing the results of a case study they performed involving 31 versions of the KornShell [24]. In that study, the predictor reported that, on average, it was expected that 87.3% of the tests would be selected. Using the **TestTube** approach, 88.1% were actually selected on average over the 31 versions. The authors explain, however, that because of the way their selective regression testing model employs averages, the accuracy of their predictor might vary significantly in practice from version to version. In later work, we present additional empirical studies to evaluate the effectiveness and accuracy of Rosenblum and Weyuker’s model [10]. Our results suggest that the distribution of modifications made to a program can play a significant role in determining the accuracy of a predictive model of test selection, and that a useful prediction model must account for both code coverage and modification distribution.

The studies of regression-test-selection techniques and test-suite-size predictors suggest that there are a number of factors that affect the cost and precision of test selection: the precision of the test-selection algorithm, the composition and size of T , and the nature of the modifications from P to P' . Based on these findings, there are a number of areas for future work, both in research and in experimentation, related to regression test selection.

3.2 Coverage Identification

The regression-test-selection techniques, described in preceding sections, identify those tests in T to rerun after modifications are made to P . However, after running the selected tests, T' , there may be parts of P' that are not tested or are not covered according to some testing criterion. Coverage-identification techniques compute test requirements for these untested or uncovered parts of P' .

Several techniques [2, 12, 19] store the data-flow testing requirements (i.e., those definition-use pairs that

were required for the original program), and incrementally update these requirements using the program modifications.⁸ Instead of storing the testing requirements for the original program, we presented a technique that uses a demand approach to transitively compute data- and control-dependences for the modifications [7]. We later presented an alternative version of our regression-test-selection algorithm that also uses a demand approach to identify definition-use pairs that should be retested, to support data-flow testing criteria [20]. Both algorithms use slicing techniques to identify the affected definition-use pairs.

To date, there has been no significant experimental evaluation of coverage-identification techniques. Thus, future work that performs such evaluations could provide evidence as to the effectiveness of these techniques in practice. Additionally, such evaluations could help to guide the development of new coverage-identification techniques..

3.3 Test-Suite Minimization

As a program evolves, new test cases may be developed and added to the test suite, causing the test suite to grow and making efficient test-suite management desirable. Test-suite-minimization techniques reduce the size of the test suite by removing obsolete and redundant tests. For example, if node coverage is required for testing P , tests $t1$ and $t4$ are sufficient; thus, tests $t2$ and $t3$ are redundant and can be eliminated.

Given test suite T , a set of test requirements R_1, R_2, \dots, R_n , that must be satisfied to provide coverage of P , and subsets of T , T_1, T_2, \dots, T_n , one associated with each of the R_i s such that any one of the tests t_j belonging to T_i can be used to test R_i , test-suite-minimization techniques find a representative set of tests from T that satisfies all of the R_i s [9, p. 272]. A representative set of test cases that satisfies all of the R_i s must contain at least one test case from each T_i ; such a set is called a *hitting set* of the group of sets T_1, T_2, \dots, T_n . To achieve a maximum reduction, it is necessary to find the smallest representative set of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the T_i s, and the problem of finding such a set is NP-complete [5]. Thus, minimization techniques resort to heuristics. Several test suite minimization heuristics have been proposed (e.g., [9, 14]). Because these heuristics are similar, a detailed discussion of them is omitted; see the papers for details.

Several recent studies examine the costs and benefits of test-suite reduction. Wong, Horgan, London, and Mathur [34] present a study that involved ten common C UNIX utility programs, including nine programs ranging in size from 90 to 289 lines of code, and one program of 842 lines of code. Graduate students injected simple mutation-like faults into each of the subject programs; faulty programs whose faults could not be detected by any tests were eliminated. All told, 183 faulty versions of the programs were retained for use in the study. The researchers minimized their test suites using ATACMIN [14], a heuristic-based minimization tool that found “exact solutions for minimizations of all test suites examined” [34, page 42].

We performed a study on a variety of 100-500 line programs; each program has a variety of versions, each containing one fault[27]. Each program also has a large universe of inputs.⁹ We performed the minimization

⁸See [29] for a thorough discussion of these techniques.

⁹These programs, versions, and inputs were created by researchers at Siemens Corporate Research, and have been used for

with an implementation of the Harrold-Gupta-Soffa minimization technique [9] using the Aristotle program-analysis system [11].

Both studies suggest that test suite minimization can result in test-suite size reduction, and both studies also suggest that fault-detection effectiveness decreases as test-suite size increases. The studies differ in their empirical results with respect to the loss of fault-detection capabilities of the reduced test suite: Wong et al.'s empirical results suggest that there is no significant loss in fault-detection capabilities of the reduced test suite whereas our results suggest that the loss of these capabilities can be significant. One possibility is that the factor likely to be most responsible for differences in results of the two studies involves the types of test suites utilized. Given this suggestion that some factor other than test-suite size influences the reduction in fault-detection effectiveness that is associated with minimization, we may not want to minimize test suites strictly in terms of code coverage. Additional experimentation and development of minimization strategies, whose cost-benefit tradeoffs are more clear, could be beneficial.

4 Areas for Future Work

The selective retest techniques and empirical evaluation of those techniques, presented in the preceding sections, suggest many areas for future work; this section discusses some of them.

Relative effectiveness and precision of regression-test-selection techniques. Although several studies [6, 22, 29, 26] have investigated regression-test-selection techniques, there is still no conclusive evidence of the relative effectiveness and efficiency of the techniques in practice. The algorithms of Ball, Rothermel and Harrold, Vokolos and Frankl, and Chen, Rosenblum, and Vo, for example, vary in the precision of the analyses performed to select the tests, and, analytically, they vary in the precision of the test suites selected. But how do the results compare in practice, and how can professional testers select the appropriate algorithm to use? The studies by Graves et al. compared the relative effectiveness of control-flow and data-flow-based approaches. But how do these techniques compare in efficiency? Empirical studies that consider software systems of varying sizes and types, evolving test suites for those systems, and real change histories can provide information that will help researchers identify the tradeoffs of using these techniques and develop guidelines that professional testers can use when making regression-testing decisions.

Generalized algorithms. The regression-test-selection techniques, described in preceding sections, use a source-code representation of the software. Can these techniques be generalized so that they apply to other formal representations of the software, such as its requirements or architecture? Our technique [30], for example, can be applied to various levels of the software if (1) the software can be represented as a graph, (2) tests can be associated with edges in this graph, and (3) the difference in two nodes in the graph can be determined. When our technique is applied to the source code, for example, the graph representation is

many studies; details of the design of the faulty versions, tests, and test suites can be found in [27].

a control-flow graph, the association between tests and edges is determined by instrumenting and executing the program to produce an edge profile, and the difference between nodes is accomplished by comparing the text of the source-code statements associated with the nodes in the graphs. Richardson, Stafford, and Wolf present a set of architecture-based coverage criteria [21]. Their criteria are based on the CHAM architecture model, and provide a type of structural coverage of the architecture. Their criteria include the all-processing-elements criterion, which requires that all processing elements are executed, and the all-connecting-elements criterion, which requires that all communication channels and connections on them are exercised. By considering the processing elements as nodes and the communication channels as edges in a graph, can we apply our regression-test-selection algorithm to the formal representation of the architecture? Test-selection algorithms applied at this level could be used in two ways. First, these algorithms could determine the retesting that is required after changes are made to the software's architecture. Applying the test selection at the architectural level may be more efficient than at the source-code level. Second, the test selection could be used to assess the retesting that would be required for differing, optional modifications to an architecture.

Hybrid algorithms. The results of the studies described in the preceding sections suggest that some type of hybrid approach to regression test selection may be beneficial. Ball, Rothermel and Harrold, Vokolos and Frankl, and Chen, Rosenblum, and Vo present algorithms that vary in the precision of the analysis and the precision of the results. Preliminary evaluation shows that these techniques also vary in the precision of the tests selected and the analysis time to select the tests. Could a less precise technique be used to get an approximate solution, and then, with input from the tester, be refined until the desired level of precision is achieved? Such an approach would let the tester tailor the precision of T' to the application or focus on critical parts of the modified software. Graves et al.'s [6] study showed that, for the subjects they studied, the tests selected by **DejaVu** and by using a data-flow-coverage approach differed very little in the faults detected or the precision of the tests selected; they did not measure the time to perform the analyses to select the tests. Could these two techniques be integrated so that the test selection would consider both control flow and data flow? Such an approach would possess the best features of each technique.

Test-suite minimization and prioritization. In some cases, after T' is selected using a test-selection algorithm, there may not be sufficient time to run all tests in T' . In this case, we may want to minimize the tests in T' according to some criteria. For example, suppose that we can approximate the time required to run the tests in T' but the time allocated for testing is not sufficient to run all tests in T' . Can we select a subset of T' for use in testing P' that could be run in the time allocated and would be effective for testing P' ? Although not safe, such a test suite should be more effective than a randomly-selected test suite. Another approach to selecting a subset of T' for use in testing P' is to prioritize the tests in T' by some criteria. For example, can we prioritize tests in T' by the coverage that they provide? Wong et al. discussed this approach in [36]. For another example, can we prioritize tests in T' by an estimation of their

fault-detection ability? Under the first approach, we should achieve coverage of P' quickly whereas under the second approach, we should expose faults early in the testing of P' .

Regression testability. *Regression testability* refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression tested. Leung and White [15] classify a program as regression testable if most single statement modifications to P entail rerunning a small proportion of T . Extending Leung and White's work, Rosenblum and Weyuker [24] consider P and T , and present a formal model of the cost-effectiveness of regression-test-selection techniques. Leung and White also discuss the *regression testability* of a software system – a system is regression testable if most single statement modifications will entail rerunning a small proportion of the current test suite [15]. Under this definition, regression testability is a function of both the design of the program and the test suite. Rosenblum and Weyuker presented a model for predicting the cost-effectiveness of regression-test-selection techniques in terms of the coverage provided by a particular test suite [23]. Under their approach, both the program and the test suite are used for prediction. Using these notions of regression testability, can we design regression testable test suites? Additionally, can we identify this testability using various representations of the software, such as its architecture? These techniques can help design software and test suites on which efficient regression testing can be performed, and the ability to consider regression testability early in the development process has the potential to provide significant savings in the cost of development and maintenance of the software.

References

- [1] T. Ball. The limit of control flow analysis for regression testing. In *International Symposium on Software Testing and Analysis*, pages 143–242, March 1998.
- [2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.
- [3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.
- [4] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.
- [5] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.
- [6] T. L. Graves, M. J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *The 20th Int'l. Conf. on Softw. Eng.*, April 1998.
- [7] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6:83–111, 1996.
- [8] M. J. Harrold. Architecture-based regression testing of evolving software. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 73–77, June 1998.
- [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] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. Technical Report OSU-CISRC-2/98-TR06, The Ohio State University, February 1998.
- [11] 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, The Ohio State University, March 1997.
- [12] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 362–367, October 1988.

- [13] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the Third Workshop on Software Testing, Analysis, and Verification*, pages 97–102. ACM Press, December 1989.
- [14] J. R. Horgan and S. A. London. ATAC: A data flow coverage testing tool for C. In *Proc. of the Symp. on Assessment of Quality Softw. Dev. Tools*, pages 2–10, May 1992.
- [15] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 60–69, October 1989.
- [16] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance '91*, pages 201–208, October 1991.
- [17] R. Lewis, D. W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *ESEC '89. 2nd European Software Engineering Conference Proceedings*, pages 487–496. Springer-Verlag, September 1989.
- [18] A. K. Onoma, W.-T. Tsai, M. H. Poonawala, and H. Sukanuma. Regression testing in an industrial environment. *communications of the ACM*, 41(5), May 1998.
- [19] T. J. Ostrand and E. J. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247, September 1988.
- [20] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.
- [21] D. J. Richardson and J. Stafford. What makes one software architecture more testable than another? In *Proceedings of the International Software Architecture Symposium*, October 1996.
- [22] D. Rosenblum and G. Rothermel. An empirical comparison of regression test selection techniques. In *Proceedings of the International Workshop for Empirical Studies of Software Maintenance*, October 1997.
- [23] D. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. In *ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, October 1996.
- [24] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Trans. Softw. Eng.*, 23(3):146–156, March 1997.
- [25] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. Ph.D. dissertation, Clemson University, May 1996.
- [26] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. on Softw. Eng.*, 24(6), June 1998.
- [27] 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. In *Proceedings of the International Conference on Software Maintenance*, November 1998.
- [28] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.
- [29] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), August 1996.
- [30] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. In *ACM Transactions on Software Engineering and Methodology*, pages 173–210, April 1997.
- [31] J. Stafford, D. J. Richardson, and A. L. Wolf. Chaining: A dependence analysis technique for software architecture. September 1997.
- [32] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.
- [33] F. Vokolos and P. Frankl. Empirical evaluation of the textual differencing regression testing techniques. In *International Conference on Software Maintenance*, November 1998.
- [34] W. Wong, R. Horgan, S. London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th Intl. Conf. Softw. Eng.*, pages 41–50, April 1995.
- [35] W. Wong, R. Horgan, S. London, and A. Mathur. A study of effective regression testing in practice. In *8th International Symposium on Software Reliability Engineering*, pages 264 – 275, November 1997.
- [36] 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. Technical Report SERC-TR-173-P, Software Engineering Research Council, April 1997.