9-1-2001

The Impact of Test Suite Granularity on the CostEffectiveness of Regression Testing

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

Alexey Malishevsky
Oregon State University

Praveen Kallakuri
University of Nebraska-Lincoln, pkallaku@cse.unl.edu

Brian Davia
Oregon State University

University of Nebraska-Lincoln, Computer Science and Engineering
Technical Report # TR-UNL-CSE-2001-0003; issued 09/01/2001

The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing

Gregg Rothermel[†], Sebastian Elbaum[‡], Alexey Malishevsky[†], Praveen Kallakuri[‡], Brian Davia[†]

[†]Department of Computer Science
Oregon State University
Corvallis, Oregon
{grother,malishal,davia}@cs.orst.edu

[‡]Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
{elbaum,pkallaku}@cse.unl.edu

ABSTRACT

Regression testing is an expensive testing process used to validate software following modifications. The cost-effectiveness of regression testing techniques varies with characteristics of test suites. One such characteristic, test suite granularity, involves the way in which test inputs are grouped into test cases within a test suite. Various cost-benefits tradeoffs have been attributed to choices of test suite granularity, but almost no research has formally examined these tradeoffs. To address this lack, we conducted several controlled experiments, examining the effects of test suite granularity on the costs and benefits of several regression testing methodologies across six releases of two non-trivial software systems. Our results expose essential tradeoffs to consider when designing test suites for use in regression testing evolving systems.

1. INTRODUCTION

As software evolves, test engineers regression test it to validate new features and detect whether new faults have been introduced into previously tested code. Regression testing is expensive, and many approaches have been suggested for lowering its cost. One approach re-uses all previously developed test cases, executing them on the modified program. When only a small portion of a system is modified, however, this *retest-all* approach can waste resources running unnecessary tests. Thus, regression test selection techniques [5, 17, 25] can be used instead to select a subset of an existing test suite. Test re-execution can also be aided by test case prioritization techniques [6, 29, 30], which order test cases so that those that are better at achieving testing objectives are run earlier in the regression testing cycle. Finally, test suite reduction techniques [4, 10, 20] can reduce testing costs by eliminating redundant test cases from test suites.

The cost-effectiveness of regression testing techniques can vary with characteristics of test suites [6, 26, 27]. One such characteristic is *test suite granularity*, which reflects the way test inputs are grouped into test cases within a test suite. For example, a test suite for a word processor might contain just a few test cases that start up the system, open a document, issue hundreds of editing commands, and close the document, or it might contain hundreds of test cases that each issue only a few commands. A test suite for a compiler might contain several test cases that each compile a source file containing hundreds of language constructs, or hundreds of test cases that each compile a source file containing just a few constructs. A test suite for a class library might contain a few test drivers that each invoke dozens of methods, or dozens of drivers that each invoke a few methods.

Faced with such choices in test suite design, test engineers may wonder which direction to take. Textbooks and popular testing literature offer advice. For example, Beizer [2, p. 51] writes: "It's better to use several simple, obvious tests than to do the job with fewer, but grander, tests." Other advice is framed in terms of test case size – an important factor in test suite granularity. Kaner et al. [13, p. 125], suggest that large tests can save time, provided they are not overly complicated, in which case simpler tests may be more efficient. Kit [15, p. 107] suggests that when testing valid inputs for which failures should be infrequent, large test cases are preferable. Bach [1] states that small test cases cause fewer difficulties with cascading errors, but large test cases are better at exposing system level failures involving interactions between software components.

Despite such suggestions, in our search of the research literature we find little formal examination of the cost-benefits tradeoffs associated with choices of test suite granularity. A more thorough investigation of these tradeoffs, and the implications of these tradeoffs on testing across the software lifecycle, could help test engineers design test suites that support more cost-effective regression testing.

We therefore designed and conducted a family of controlled experiments, in which we observed the application of test suites of various granularities across six releases of two non-trivial software systems. We measured the impact of test suite granularity on the costs and savings of several regression-testing methodologies: retest-all, regression test selection, test case prioritization, and test suite reduction. Our results expose essential tradeoffs to consider when designing test cases for use in regression testing evolving software systems.

2. OVERVIEW AND RELATED WORK

Following Binder [3], we define a *test case* to consist of a pretest state of the system under test (including its environment), a sequence of test inputs, and the expected test results. We define a *test suite* to be a set of test cases.

A definition of *test suite granularity* is harder to come by, but the testing problem we are addressing is a practical one, so we begin by drawing on examples. Test engineers designing test cases for a system identify various *testing requirements* for that system, such as specification items, code elements, or method sequences. Next, they must construct test cases that *exercise* these requirements. An engineer testing a word processor might specify sequences of editing commands, an engineer testing a compiler might create sample target-language programs, and an engineer testing a class library might develop drivers that invoke methods. The practical questions these engineers face include: how many editing commands to include per sequence, how many constructs to include in each target-language program, and how many methods to invoke per driver.

The answers to these questions involve many cost-benefits tradeoffs. For example, if the cost of performing setup activities for individual test cases dominates the cost of executing those tests, a test suite containing a few large test cases can be less expensive than a suite containing many small test cases. Large test cases might also be better than small ones at exposing failures caused by interactions among system functions. Small test cases, on the other hand, can be easier to use in debugging than large test cases, because they reduce occurrences of cascading errors [1] and simplify fault localization [11]. Further, grouping test inputs into large test cases may prevent test inputs that appear later in the test cases from exercising the requirements they are intended to exercise, by causing later inputs to be applied from system states other than those intended.

In part, the foregoing examples involve *test case size*, a term used informally in [1, 2, 13, 15] to denote notions such as the number of commands applied to, or the amount of input processed by, the program under test, for a given test case. However, there is more than just test case size involved: when engineers increase or decrease the number of requirements covered by each test case, this directly determines the number of individual test cases that need to be created to cover all the requirements. It is the interaction of test case size and number of test cases that creates most of the cost-benefits tradeoffs just discussed.

The phenomenon we wish to study in this paper, then, involves ways in which, in the course of designing a test suite to cover requirements, test inputs are grouped into test cases within that suite. Thus, we use the term *test suite granularity* to describe a partition on a set of test inputs into a test suite containing test cases of a given size. Section *3.1.1.2* provides a more precise measure of test suite granularity.

Other definitions of test case, test case size, and test suite granularity are possible. Test engineers might choose to view the individual inputs applied during a single invocation of a word processor, or the individual method invocations made from within a class driver, as individual test cases, each with its own size. Also, in practice test suites may contain test cases of varying size. However, our definitions facilitate the controlled study of the cost-benefits tradeoffs outlined above. Moreover, our definitions are suitable for use with the programs we use as subjects in our experiments.

2.1 Regression Testing and Methodologies

We wish to study the effects of test suite granularity on the costs and effectiveness of testing activities across the software lifecycle, i.e., in relation to regression testing.

Let P be a program, P' be a modified version of P, and T be a test suite developed for P. Regression testing seeks to test P'. To facilitate such testing, test engineers may re-use T to the extent possible, but new test cases may also be required to test new functionality. Both reuse of T and creation of new test cases are important; however, it is test reuse that concerns us here, as it is test reuse that motivates most suggestions about costs/benefits of test case size.

In particular, we consider four methodologies related to regression testing and test reuse: retest-all, regression test selection, test suite reduction, and test case prioritization.

2.1.1 Retest-all

When P is modified, creating P', test engineers may simply reuse all non-obsolete[1] test cases in T to test P'; this is known as the *retest-all* technique [16]. The retest-all technique represents typical current practice [21], and thus, serves as our *control technique.*

2.1.2 Regression Test Selection

The *retest all* technique can be expensive: rerunning all test cases may require an unacceptable amount of time or human effort. *Regression test selection* (RTS) techniques [5, 17, 25] use information about P, P', and T to select a subset of T with which to test P'. (For a survey of RTS techniques, see [24].) Empirical studies of some RTS techniques [5, 8, 26, 28] have shown that they can be cost-effective.

One cost-benefits tradeoff among RTS techniques involves *safety* and *efficiency.* Safe RTS techniques guarantee that, under certain conditions, test cases not selected could not have exposed faults in P' [24]. Achieving safety, however, may require inclusion of a larger number of test cases than can be run in available testing time. Non-safe RTS techniques sacrifice safety for efficiency, selecting test cases that, in some sense, are more useful than those excluded.

2.1.3 Test Suite Reduction

As P evolves, new test cases may be added to T to validate new functionality. Over time, T grows, and its test cases become redundant in terms of code or functionality exercised. *Test suite reduction* techniques[2] [4, 10, 20] address this problem by using information about P and T to permanently remove redundant test cases from T, so that subsequent reuse of T can be more efficient. Test suite reduction thus differs from regression test selection in that the latter does not permanently remove test cases from T, but simply "screens" those test cases for use on a specific version P' of P, retaining unused test cases for use on later releases.

By reducing test-suite size, test-suite reduction techniques reduce the costs of executing, validating, and managing test suites over future releases of the software. A potential drawback of test-suite reduction, however, is that removal of

[1]Test cases in T that no longer apply to P' are *obsolete*, and must be reformulated or discarded [16].

[2]Test suite reduction has also been referred to, in the literature, as *test suite minimization*; however, the intractability of the test suite minimization problem forces techniques to employ heuristics that may not yield minimum test suites; hence, we term these techniques "reduction" techniques.

test cases from a test suite may damage that test suite's fault-detecting capabilities. Some studies [27] have shown that test suite reduction can significantly reduce the fault-detection effectiveness of test suites. Other studies [31] have shown that test-suite reduction can produce substantial savings at little cost to fault-detection effectiveness.

2.1.4 Test Case Prioritization

Test case prioritization techniques [6, 29, 30] schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or increases the likelihood that those test cases will detect faults early in testing.

Empirical results [6, 29, 30] suggest that several simple prioritization techniques can significantly improve one testing performance goal: namely, the rate at which test suites detect faults. An improved rate of fault detection during regression testing provides earlier feedback on the system under test and lets software engineers begin locating and correcting faults earlier than might otherwise be possible.

2.2 Related Work

Many papers have examined the costs and benefits of regression test selection, test case prioritization, and test case reduction [5, 6, 8, 14, 27, 31]. Several textbooks and articles on testing [1, 2, 6, 11, 13, 15, 27] have discussed tradeoffs involving test suite granularity. None of these documents, however, has formally examined these tradeoffs, or done so in relation to regression testing.

In [26, 28], test suite granularity is specifically considered as a factor in two studies of regression test selection, and test suites constructed from smaller test cases are shown to facilitate selection. These studies, however, measured only numbers of test cases selected, and considered only safe RTS techniques. In contrast, this paper presents the results of a controlled experiment designed specifically to examine the impact of test suite granularity on the costs and savings associated with several regression testing methodologies, across several metrics of importance.

3. THE EXPERIMENT

Informally, the research question we address is, "how does test suite granularity affect the costs and benefits of regression testing methodologies across software release cycles?" More formally, we wish to evaluate the following hypotheses (expressed as null hypotheses) for three methodologies — regression test selection, test suite reduction, and test case prioritization — at a 0.05 level of significance:

H1 (test suite granularity): Test suite granularity does not have a significant impact on the costs and benefits of regression testing techniques.

H2 (technique): Regression testing techniques do not perform significantly differently in terms of the selected costs and benefits measures.[3]

H3 (interaction): Test suite granularity effects across regression testing techniques do not significantly differ.

[3]This hypothesis has been tested in previous studies, and is included primarily for completeness and replication.

We also wished to evaluate these hypotheses relative to the retest-all technique. To simplify this, we treated that technique as a control technique and assessed it in the context of each of the other three methodologies.

To test our hypotheses we designed several controlled experiments. The following sections present our measures, materials, design, threats to validity, and results.

3.1 Measures

3.1.1 Independent Variables

Our experiments manipulated two independent variables: regression testing technique and test suite granularity.

3.1.1.1 Regression testing techniques.

For each regression testing methodology considered we studied two or three techniques. In selecting techniques we had two goals: (1) to include techniques that could serve as practical experimental controls, and (2) to include techniques that could easily be implemented by practitioners.

Regression test selection. We chose three RTS techniques, *retest-all*, *modified entity*, and *modified non-core entity*. As described in Section 2.1, the retest-all technique [16] executes every test case in T on P', and is our control technique, representing typical current practice. The *modified entity* technique [5] is a *safe* RTS technique: it selects test cases that exercise functions, in P, that (1) have been deleted or changed in producing P', or (2) use variables or structures that have been deleted or changed in producing P'. The *modified non-core entity* technique works like the modified-entity technique, but ignores "core" functions, defined as functions exercised by more than k% of the test cases in the test suite (we set k to 80%). This technique trades safety for savings in re-testing effort (selecting all test cases through core functions may lead to selecting all of T).

Test suite reduction. We selected two test suite reduction techniques, *no reduction* and *GHS reduction*. The no reduction technique (equivalent to retest-all) represents current typical practice and acts as our control. The GHS reduction technique is a heuristic presented by Gupta, Harrold, and Soffa [10] that attempts to produce suites that are minimal for a given coverage criterion; we used a function coverage criterion.

Test case prioritization. We selected three test case prioritization techniques, *random* prioritization, *additional function coverage* prioritization, and *optimal* prioritization. Random prioritization (equivalent to retest-all in our experiments, because we randomly ordered the test cases in our test suites before using them) places test cases in T in random order and is our control. Additional function coverage prioritization [29] iteratively selects a test case that yields the greatest function coverage, then adjusts the coverage information on subsequent test cases to indicate their coverage of functions not yet covered, and then repeats this process, until all functions covered by at least one test case have been covered. When all functions have been covered, this process is repeated on the remaining test cases. Optimal prioritization uses information on which test cases in T reveal faults in P' to find an (approximate) optimal ordering for T; though not a practical technique (in practice we don't know which test cases reveal which faults beforehand), this technique provides an upper bound on prioritization benefits.

3.1.1.2 *Test suite granularity.*

Our objective was to quantify the impact of varying test suite granularities on the costs and benefits of regression testing techniques. To do this, we needed to obtain test suites of varying granularities, in a manner that controls for other factors that might affect our dependent measures. We considered two approaches for doing this.

The first approach is to obtain or construct test suites for a program, partition them into subsets according to size, and compare the results of executing these different subsets. However, this approach would not let us establish a causal relationship between test suite granularity and measures of costs or benefits, because it does not control for other factors that might influence those measures.

To see this, suppose T can be partitioned into subsets T_1 containing test cases of size s, and T_2 containing test cases of size ks. Suppose we compare costs or benefits of T_1 and T_2 and find that they differ. In this case, we cannot determine whether this difference was caused by test suite granularity, or by differences in the number or type of inputs applied in T_1 and T_2. (For example, it might be the case that all modified functions are exercised only by inputs in T_1.)

The second approach we considered is to construct test suites of varying granularities by sampling a single pool or "universe" of *test grains*. A *test grain* is a smallest input that could be used as a test case (applied from a start state and producing a checkable output) for a target program. A *sampling procedure* can randomly select test grains to create test cases of different sizes: a test case of size s consists of s test grains. Applying this sampling procedure repeatedly to a universe of n test grains, without replacement, until none remain (partitioning the universe into n/s test cases of size s, and possibly one smaller test case), yields a test suite of *granularity level* s. Repeating this procedure for various values of s gives us test suites of different granularity levels that can be compared controlling for differences in types and numbers of inputs.

We chose the second approach, and employed four granularity levels: 1, 4, 16 and 64, which we refer to as G1, G4, G16, and G64, respectively.

3.1.2 *Dependent Variables*

To investigate our hypotheses we need to measure the costs and benefits of regression testing techniques. To do this we constructed three models. Our first two models assess costs and benefits of regression test selection and test case reduction, and our third model assesses the benefits of test case prioritization. We restrict our attention to the costs and benefits measured by these models; other costs and benefits are mentioned in Section 3.4.

3.1.2.1 *Savings in test execution time.*

Regression test selection and test suite reduction achieve savings by reducing the number of test cases that need to be executed on P', thereby reducing the effort required to retest P'. The use of larger test suite granularities is also expected to produce savings in test execution and validation time. In this experiment, to evaluate these savings, we measure the time required to execute and validate[4] the test cases in test suites, selected test suites, and reduced test suites, across test suites of different granularities.

[4]Validation involved using the Unix "diff" utility to compare all old and new outputs and external files.

3.1.2.2 *Costs in fault-detection effectiveness.*

One potential cost of regression test selection and test suite reduction is the cost of missing faults that would have been exposed by excluded test cases. This cost could also vary with test suite granularity. Costs in fault-detection effectiveness can be measured by studying programs that contain known faults. When dealing with single faults, one common measure [8, 12] estimates whether a test case t detects fault f in P' by applying t to two versions of P', one that contains f and one that does not. If the outputs of P and P' differ on t, we conclude that t reveals f.

In our experiments, however, we wish to study programs containing multiple faults. When P' contains multiple faults it is not enough to note which test cases cause P' to fail, we must also determine which test cases could contribute to revealing which faults. One way to do this [14] is to instrument P' such that when t is run on P' we can determine, for each fault f in P', whether: (1) t reaches f, (2) t causes a change in data state following execution of f, and (3) the output of P' on t differs from the output of P on t.

One drawback of this approach is that it can underestimate the faults that could be found in practice with t. For example, suppose P' contains faults f_1 and f_2, which can each be detected by t if present alone. Suppose, however, that when f_1 and f_2 are both present in P', f_1 prevents t from reaching f_2. This approach would suggest that t cannot detect f_2. In a debugging process, however, an engineer might detect and correct f_1, and then on re-running t on the (partially) corrected P', be able to detect f_2. A second drawback of this approach is that testing for data state changes can be infeasible in programs that manipulate enormous data spaces, such as those used in this study.

For these reasons, we chose a second approach. We activated each fault f in P' individually, executed each test case t (at each granularity level) on P', and determined whether t detects f singly by noting whether it causes P and P' to produce different outputs. We then assumed that detection of f when present singly implies detection of f when present in combination with other faults.

This approach avoids the drawbacks of the first approach: it accommodates incremental fault-correction and doesn't require detection of data state changes. However, the approach may err in cases where multiple faults would mask each other's effects such that no failures would occur on t. We investigated the possible magnitude of this error by also executing our test cases on our multi-fault versions, and measuring the extent to which test cases that caused single-fault versions to fail did not cause multi-fault versions to fail.[5] The data showed that for one of these programs (*emp-server*, described momentarily), across all versions and granularities, masking occurred on only 16 of 13,195 test cases (.12%), and for the other program (*bash*), across all versions and granularities, masking occurred on only 240 of 7,760 test cases (3.09%).

3.1.2.3 *Savings in rate of fault detection.*

The test case prioritization techniques we consider have a goal of increasing a test suite's rate of fault detection.

[5]This check does not eliminate the possibility that some subset of the faults in a multi-fault version might mask one another, and be undetected by test case t in that version even though detected singly by t; however, it was not computationally feasible to check for this possibility.

Program	*Version*	*Functions*	*Changed Functions*	*Lines of Code*
emp-server	4.2.0	1,159	—	67,719
emp-server	4.2.1	1,159	51	67,719
emp-server	4.2.3	1,171	286	68,626
emp-server	4.2.4	1,172	9	69,796
emp-server	4.2.5	1,173	100	68,739
emp-server	4.2.6	1,173	31	68,782
bash	2.0	1,499	—	48,292
bash	2.01	1,541	303	49,555
bash	2.01.1	1,542	39	49,666
bash	2.02	1,683	519	58,090
bash	2.02.1	1,683	12	58,103
bash	2.03	1,712	196	59,010

Table 1: Experiment Subjects.

To measure rate of fault detection, in [29] we introduced a metric, *APFD*, which measures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates. More formally, let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T' of T which reveals fault i. The APFD for test suite T' is given by the equation:

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

Examples and empirical results illustrating the use of this metric are provided in [29].

3.2 Experiment Materials

3.2.1 Programs

For these experiments we obtained several releases of two non-trivial C programs: `emp-server` and `bash`. Table 1 provides data on these programs, we describe that data below.

3.2.1.1 Emp-server program characteristics.

`Emp-server` is the server component of the open-source client-server internet game `Empire`. `Emp-server` is essentially a transaction manager: its main routine consists of initialization code followed by an event loop in which execution waits for receipt of a user command. `Emp-server` is invoked and left running on a host system; a user communicates with the server by executing a `client` that transmits the user's inputs as commands to `emp-server`. When `emp-server` receives a command, its event loop invokes routines that process the command, then waits to receive the next command. As `emp-server` processes commands, it may return data to the client program for display on the user's terminal, or write data to a local database (a directory of ASCII and binary files) that keeps track of game state. The event loop and program terminate when a user issues a "quit" command.

We obtained six versions of `emp-server`. Table 1 shows the numbers of functions and lines of executable code in each version, and for each version after the first, the number of functions changed for that version (modified or added to the version, or deleted from the preceding version).

3.2.1.2 Bash program characteristics.

`Bash` [23], short for "Bourne Again SHell", is a popular open-source application that provides a command line interface to multiple Unix services. `Bash` was developed as part of the GNU Project, adopting several features from the Korn and C shells, but also incorporating new functionality such as improved command line editing, unlimited size command history, job control, indexed arrays of unlimited size, and more advanced integer arithmetic.

`Bash` is still evolving; on average two new releases have emerged per year over the last five years. For this experiment we used six versions of `bash` released from 1996 to 1999 (see Table 1). Each release corrects faults, but also provides new functionality as evident by the increasing code size.

	`emp-server`	`bash`
G1	1985	1168
G4	497	292
G16	125	73
G64	32	19

Table 2: Test Cases per Granularity Level.

3.2.2 Test Cases and Test Automation

To examine our research question we required test cases for our subject programs. Moreover, these test cases had to be structured in a way that facilitates the controlled investigation of the effects of test suite granularity, following the methodology outlined in Section *3.1.1.2*. The approaches used to create and automate these test cases, which differed somewhat between our subject programs, were as follows.

3.2.2.1 Emp-server test cases and test automation.

No test cases were available for `emp-server`. To construct test cases we used the `Empire` information files, which describe the 196 commands recognized by `emp-server`, and the parameters and environmental effects associated with each. We treated these files as informal specifications for system functions and used them, together with the category partition method [22], to construct a suite of test cases for `emp-server` that exercise each parameter, environmental effect, and erroneous condition described in the files.

We deliberately created the smallest test cases possible, each using the minimum number of commands necessary to cover its target requirement. Each test case consists of a sequence of between one and six lines of characters (average 1.2 lines per test case), and constitutes a sequence of inputs to the `client`, which the `client` passes to `emp-server`. Because the complexity of commands, parameters, and effects varies widely across the various `Empire` commands, this process yielded between one and 38 test cases for each command, and ultimately produced 1985 test cases. These test cases constituted our test grains, as well as our test cases at granularity level G1. We then used the sampling procedure described in Section *3.1.1.2* to create test suites of granularity levels G4, G16, and G64, as shown in Table 2.

To execute and validate test cases automatically, we created test scripts. Given test suite T, for each test case t in T these scripts: (1) initialize the Empire database to a start state; (2) invoke `emp-server`; (3) invoke a `client` and issue the sequence of inputs that constitutes the test case to the client, saving all output returned to the client for use in validation; (4) terminate the client; (5) shut down `emp-server`; (6) save the contents of the database for use in validation; and (7) compare saved client output and database contents with those archived for the previous version. By design, this process lets us apply (in step 3) all of the test inputs contained in a test case, at all granularity levels.

3.2.2.2 *Bash test cases and test automation.*

Each version of `bash` was released with a test suite, composed of test cases from previous versions and new test cases designed to validate added functionality. We could not directly use these suites for our experiment, however, because they were composed strictly of large test cases, each exercising whole functional components. Further, the test suites executed, on average, only 33% of the functions in `bash`.

We overcame these limitations by creating a new regression test suite using two complementary methods. First, we partitioned each large test case that came with `bash` release 2.0 into the smallest possible test cases. (We used the test cases from release 2.0 because they are the only ones that work across all releases.) Second, to exercise functionality not covered by the original test suite, we created additional small test cases by considering the reference documentation for `bash` [23] as an informal specification.

The resulting test suite has 1168 test cases, exercising an average of 64% of the functions across all the versions. Each test case in the new test suite contains between one and 54 lines. Each line constitutes an instruction consisting of `bash` or `Expect` [18] commands that can be executed on an instance of `bash`. The 1168 test cases constituted our test grains, and test cases at granularity level G1. As with `emp-server`, we then followed the procedure described in Section *3.1.1.2* to create test suites at granularity levels G4, G16, and G64, as reported in Table 2.

3.2.3 *Faults*

We wished to evaluate the performance of regression testing techniques with respect to detection of regression faults – faults created in a program version as a result of the modifications that produced that version. To obtain such faults for `emp-server` and `bash`, we asked several graduate and undergraduate computer science students, each with at least two years experience programming in C and unacquainted with the details of this study, to become familiar with the programs and to insert regression faults into the program versions. These fault seeders were instructed to insert faults that were as realistic as possible based on their experience with real programs, and that involved code deleted from, inserted into, or modified in the versions.

Given ten potential faults seeded in each version of each program, we activated these faults individually, and executed the test suites (at all granularity levels) for the programs to determine which faults could be revealed by which test cases. We excluded any potential faults that were not detected by any test cases at any granularity level: such faults are meaningless to our measures and cannot influence our results. We also excluded any faults that, at *every* granularity level, were detected by more than 80% of the test cases; our assumption was that such easily detected faults would be detected by test engineers during their unit testing of modifications (only five faults fell into this category). The numbers of faults remaining after exclusions, and utilized in the studies, are reported in Table 3.

3.2.4 *Additional Instrumentation*

To perform our experiments we required additional instrumentation. Our test coverage and control-flow graph information was provided by the Aristotle program analysis system [9] and by the Clic instrumentor and monitor [7]. We created test case prioritization, test suite reduction, and regression test selection tools implementing the techniques described in Section *3.1.1.1*. We used Unix utilities and direct inspection to determine modified functions, or functions using modified structures.

Program	*Version*	*Regression Faults*
emp-server	4.2.0	–
emp-server	4.2.1	10
emp-server	4.2.3	10
emp-server	4.2.4	10
emp-server	4.2.5	10
emp-server	4.2.6	9
bash	2.0	–
bash	2.01	10
bash	2.01.1	9
bash	2.02	9
bash	2.02.1	4
bash	2.03	9

Table 3: Faults per Subject Version.

All `emp-server` timing-related data was gathered on a Sun Microsystems Ultra 10 with 256 MB of memory. All `bash` timing-related data was gathered on a SunUltra 60 with 512 MB of memory. While timing data was being collected, our testing processes were the only active user processes on the machines.

3.3 Experiment Design

To address our hypotheses, we designed three sets of experiments with the same format. Each experiment evaluates the hypotheses for regression test selection, test suite reduction, and test case prioritization. In addition, each experiment employs blocking and two factors (one for each independent variable) with multiple levels to ensure unbiased treatment assignment. Table 4 depicts our Randomized Block Factorial (RBF-22) experiment design, showing how treatment combinations were applied to subjects. The RBF-22 design let us investigate the behavior of various techniques (TQ) under different test suite granularity levels (G), and let us quantify the impact of test suite granularity on the costs and benefits of different regression testing techniques as measured by our dependent variables.

Blocking. The decision to employ program as a blocking criterion was not obvious because it could have been considered a factor or a block. However, given that our hypotheses aim exclusively at evaluating techniques across different test suite granularities, we decided to consider program a block. Since we have two programs, we use two blocks under the assumption that the observations within each program will be more homogeneous than the entire sample set. This use of program as a blocking factor minimizes the impact of program variation on experimental error.

Sample Size. To choose sample size we performed an a-priori study. Given the effort involved in preparing subject versions (in retrospect, over 80 hours per version after establishment of the initial infrastructure) we wanted to detect meaningful effects with a minimal number of invested resources. There are several statistical approaches for determining sample size [19], they differ in terms of the information they require as input for sample size calculation. We decided to determine sample size using an approximation of the difference that is worth detecting in the dependent variables (also known as "D") to distinguish practical dif-

	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.
Block `bash`	TQ_1G_1	TQ_1G_4	TQ_1G_{16}	TQ_1G_{64}	TQ_2G_1	TQ_2G_4	TQ_2G_{16}	TQ_2G_{64}
Block `emp-server`	TQ_1G_1	TQ_1G_4	TQ_1G_{16}	TQ_1G_{64}	TQ_2G_1	TQ_2G_4	TQ_2G_{16}	TQ_2G_{64}

Table 4: RBF-22 Design (TQ Stands for Technique, and G for Test Suite Granularity Level).

ferences among means. This procedure requires an estimate of the standard deviation from the population and the size of difference between means that would be worth detecting.

From previous studies on regression test selection [8], we estimated the standard deviation of the population for a subset of `emp-server`.[6] We decided that a difference of size 5% between two treatments on any of the metrics would be a meaningful difference. Next, using the degrees of freedom of the numerator and the degrees of freedom of the error term over the operating characteristic curve, we estimated that five observations per cell would be sufficient to achieve a power greater than .80 (probability of rejecting a false null hypothesis) for a two block factorial design with two treatments, and alpha=0.05 [19]. Hence, each cell in Table 4 has five observations, corresponding to five versions from each program under each treatment combination. These versions constitute random effects that we do not control, and we consider them samples from a population of versions.

3.4 Threats to Validity

In this section we describe the internal, external, construct, and conclusion threats to the validity of our experiments, and the approaches we used to limit their impact.

Internal Validity. To test our hypotheses we had to conduct a set of experiments that required a large number of processes and tools. Some of these processes involved programmers (e.g., fault seeding) and some of the tools were specifically developed for this experiment, all of which could have added variability to our results increasing the threats to internal validity. We adopted several procedures and tests to control and minimize these sources of variation. For example, the fault seeding process was performed following a specification so that every programmer operated in a similar way, and it was performed in two locations using different groups of programmers. Also, we carefully validated new tools by testing them on small sample programs and test suites, refining them as we targeted the larger subjects, and cross validating them across labs.

Having only one test suite at each granularity level in each subject might be another threat to internal validity. Although multiple test suites would have been ideal, our procedure for generating coarser granularity test suites involved randomly selecting and joining test grains, which reduces the chances of bias caused by test suite composition.

External Validity. Two issues limit the generalization of our results. The first issue is the quantity and quality of subjects. Although using only two subjects might lessen the external validity of the study, the relatively consistent results for `bash` and `emp-server` suggest that the results may generalize. Regarding the quality of the subjects, there is a large population of C programs of similar size. For example, the linux RedHat 7.1 distribution includes source code for 394 applications; the average size of these applications is 22,104 non-comment lines of code, and 19% have sizes between 25 and 75 KLOC. Still, replication of these studies on other subjects could increase the confidence in our results.

The second limiting factor is test process representativeness. Although the random grouping procedure we employed to obtain coarser granularity test suites is powerful in terms of control, it constitutes a simulation of the testing procedures used in industry, which might also impact the generalization of the results. Complementing these controlled experiments with case studies on industrial test suites, though sacrificing internal validity, could help.

Construct Validity. The three dependent measures that we have considered are not the only possible measures of the costs and benefits of regression testing methodologies. Our measures ignore the human costs that can be involved in executing and managing test suites. Our measures do not consider debugging costs such as the difficulty of fault localization, which could favor small granularity test suites [11]. Our measures also ignore the analysis time required to select or prioritize test cases, or reduce test suites. Previous work [27, 28, 29], however, has shown that — at least for the techniques considered — analysis time is either much smaller than test execution time, or analysis can be accomplished automatically and in off-hours prior to the critical regression testing period.

Conclusion Validity. The number of programs and versions we considered was large enough to show significance for some of the techniques we studied, but not for others. Although the use of more versions would have increased the power of the experiment, the average cost of preparing each version exceeded 80 hours, limiting our ability to make additional observations.

3.5 Data and Analysis

In the following sections we investigate the effects of test suite granularity on our three regression testing methodologies, in turn, employing descriptive and inferential statistics.

3.5.1 Granularity and Regression Test Selection

We begin by exploring the impact of test suite granularity on regression test selection techniques. To facilitate comparison with the control technique and save space, we present several graphs as part of Figure 1. The pair of graphs in the leftmost column present results for the retest-all technique, the pair second from left present results for the modified entity RTS technique, and the pair third from left present results for the modified non-core entity RTS technique. (The pair in the rightmost column present results for test suite reduction, discussed in the next section.)

In each graph, the horizontal axis represents test suite granularity, and the vertical axis represents either fault detection effectiveness (top row of graphs) or test execution time (bottom row of graphs). Each graph contains four data points per program, with each point representing the aver-

[6]Since we did not have any other studies on these subjects, we assumed that prioritization and reduction behaved similarly for `emp-server` and `bash`.

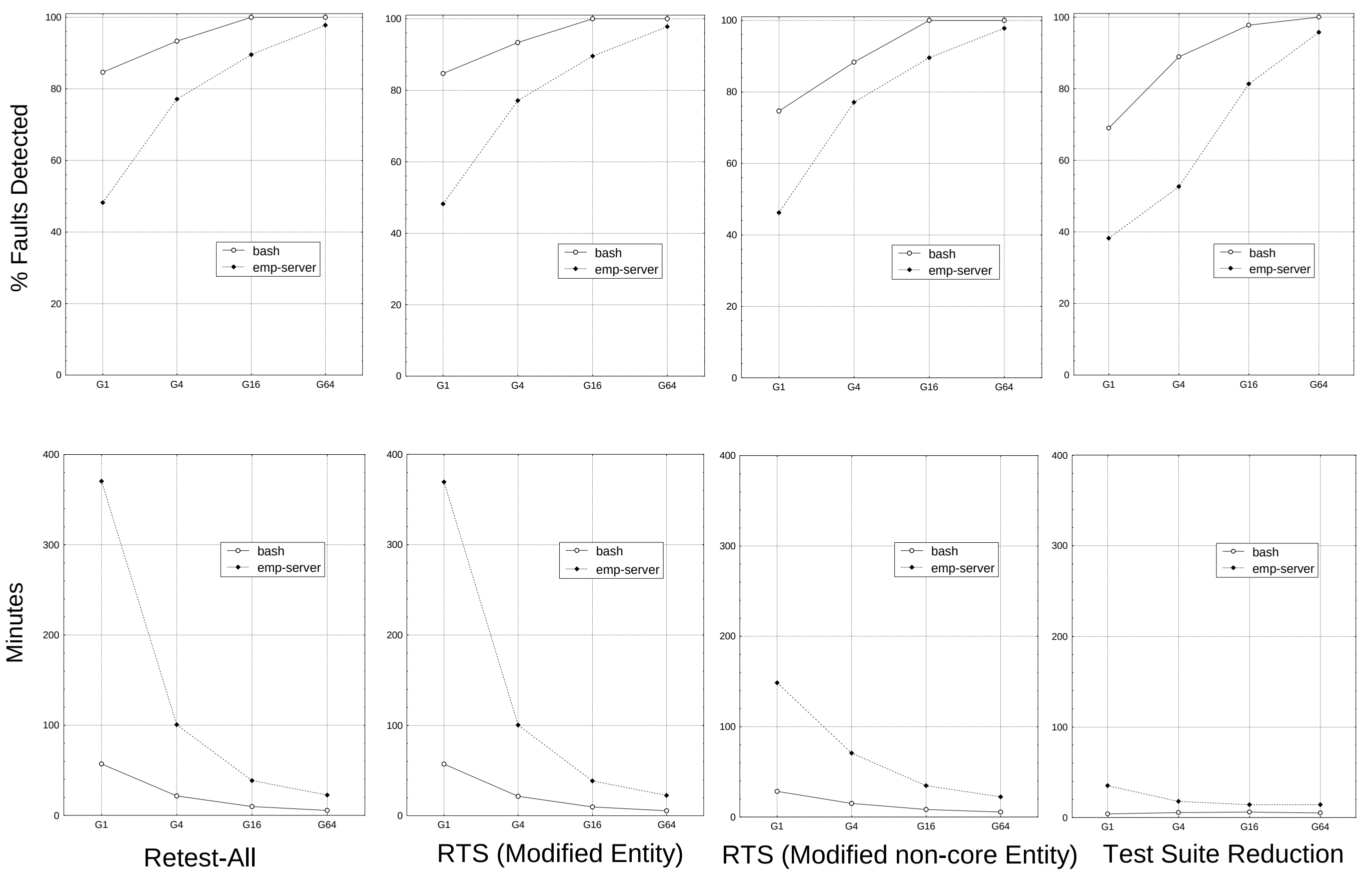

Figure 1: Fault detection effectiveness (top) and test execution time (bottom) for regression test selection and test suite reduction techniques across test suite granularities, averaged across versions.

age, across all five modified versions of the given program, of the metric being graphed (fault detection effectiveness or test execution time). We joined the data points with lines to assist interpretation.

For example, the upper-left graph shows that for the retest-all technique, fault detection effectiveness presented similar trends across test suite granularities for the two programs, increasing with granularity. For `emp-server` this increase ranged from 48% at granularity level G1 to 98% at granularity level G64, whereas for `bash` it ranged only from 85% to 100%. The lower-left graph shows that for the retest-all technique, test execution times presented similar trends, for `emp-server` ranging from 371 minutes at level G1 to 23 minutes at level G64, and for `bash` ranging from 57 minutes at G1 to 6 minutes at G64.

The trends observed across granularities for the modified-entity and modified non-core entity RTS techniques were similar to those observed for the retest-all technique; however, the two RTS techniques behaved quite differently. The modified-entity technique retained the fault-detection effectiveness of the retest-all technique; but it achieved no savings in execution on `bash`, and saved less than a minute in execution time at granularity level G1 (too small to be visible on the graphs) on `emp-server`.

In contrast to the modified entity technique, the modified non-core entity technique achieved substantial savings in test execution time on both programs, at lower test suite granularities. These savings decreased, however, as granularity increased, and were barely noticeable at granularity level G64. Notably, the modified non-core entity technique was nearly equivalent, in terms of fault-detection effectiveness, to the other two techniques; it missed two faults for test cases at granularity level G1 and one fault for test cases at level G4 in one version of bash, and only one fault in one version of `emp-server`, at granularity level G1.

To determine whether the impact of test suite granularity on our dependent variables was statistically significant, we performed an analysis of variance (Anova) on the data. Table 5 presents the results of this analysis applied to the retest-all and modified non-core entity techniques.[7] For each dependent variable we performed one independent analysis that includes the sources of variation considered, the sum of squares, degrees of freedom, mean squares, F value, and p-value for each source. Since we set alpha to 0.05, and the p-value represents the smallest level of significance that would lead to the rejection of the null hypothesis, we reject the hypothesis when p is less than alpha.

[7]Our sample size was chosen to allow us to compare pairs of techniques, and we compared each pair. However, since the data for retest-all and the modified entity technique were nearly identical, we present only the comparison between the retest-all and modified non-core entity techniques. Results of the other Anovas are available in the Appendix.

Techniques: Modified non-core Entity and Retest-all					
Variable: *Percentage of faults detected.*					
Source	SS	D.F.	MS	F	p
Granularity	15078.22	3	5026.07	19.92	0.00
Technique	90.31	1	90.31	0.36	0.55
Interaction	120.94	3	40.31	0.16	0.92
Error	18164.85	72	252.29		
Variable: *Percentage of time saved.*					
Source	SS	D.F.	MS	F	p
Granularity	7955.33	3	2651.78	8.22	0.00
Technique	11567.93	1	11567.93	35.87	0.00
Interaction	7955.33	3	2651.78	8.22	0.00
Error	23217.09	72	322.46		

Table 5: Selection Anovas.

The results indicate that test suite granularity significantly impacted fault-detection effectiveness. They also show that test suite granularity and technique can significantly impact savings in execution time. There is also a significant interaction between technique and test suite granularity when evaluating savings, which was expected given that the retest-all technique produced no savings while the modified non-core entity technique saved up to 94% of test suite execution time (at granularity level G1 on `emp-server`). These findings agree with our previous observations and conjectures. However, one place we expected significance and did not find it was in the interaction between technique and test suite granularity on fault detection effectiveness.

3.5.2 Granularity and Test Suite Reduction

Test suite reduction results were similar to those produced by regression test selection when exposed to the spectrum of test suite granularities. In the top graph in the rightmost column in Figure 1 we again observe similar patterns. In both `bash` and `emp-server`, fault-detection effectiveness increased as test suite granularity increased, in a manner similar to that observed for the retest-all technique. For `emp-server` the increase ranged from 38% at granularity level G1 to 96% at granularity level G64, whereas for `bash` the increase ranged from 89% to 100%. In the bottom graph in the rightmost column, we also see that savings in test suite execution time decreased as test suite granularity increased. For example, test suite reduction for `bash` reduced execution time by 93% at granularity level G1, but by only 10% at granularity level G64. It is apparent that as test suite granularity increased, the effectiveness of the reduced test suite increased, but the opportunities to save through reduction also decreased.

We performed an Anova to further evaluate our conjectures and test our hypotheses. Table 6, which follows the same structure as the table for regression test selection, presents the results for each dependent measure. The results indicate that granularity significantly affected both dependent measures. In addition, and differing from the findings for RTS techniques, the use of reduction significantly decreased the number of faults detected.

3.5.3 Granularity and Test Case Prioritization

Our third experiment considered test case prioritization. Within this methodology we analyze three techniques: random prioritization (through retest-all) as a control, and optimal and additional function coverage prioritization.

Figure 2 displays three graphs, one per technique, with our measure of rate of fault detection, APFD, in the y-axis.

Techniques: Reduction and Retest-all					
Variable: *Percentage of faults detected.*					
Source	SS	D.F.	MS	F	p
Granularity	17413.01	3	5804.34	20.30	0.00
Technique	1402.81	1	1402.81	4.91	0.03
Interaction	605.23	3	201.74	0.71	0.55
Error	20589.91	72	285.97		
Variable: *Percentage of time saved.*					
Source	SS	D.F.	MS	F	p
Granularity	13468.92	3	4489.64	79.49	0.00
Technique	74246.06	1	74246.06	1314.52	0.00
Interaction	13468.92	3	4489.64	79.49	0.00
Error	4066.67	72	56.48		

Table 6: Reduction Anovas.

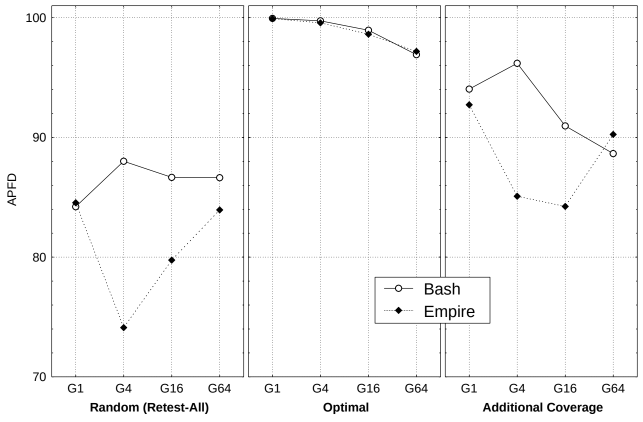

Figure 2: APFD for test case prioritization.

Results for both programs were similar under the optimal technique: there was a consistent decrease in APFD as test suite granularity increased. This was what we expected, since having more test cases provides more opportunities for prioritization; still, the differences were small. The random and additional functional coverage techniques, however, presented great variation in results that cannot be explained based solely on the increase in granularity.

The Anova presented in Table 7 confirms these observations relative to optimal and random techniques. (Anovas for all pairs of techniques are given in the Appendix; we omit the other two here because they are similar to the one in Table 7.) The techniques are significantly different. However, we could not reject the hypotheses about test suite granularity, or about interaction between techniques and granularity, and their lack of influence on variations in APFD.

4. DISCUSSION

Our results strongly support our hypothesis that test suite granularity significantly impacts the cost-effectiveness of regression testing methodologies (at least, for regression test selection and test suite reduction methodologies). In other words: granularity matters. We also rejected our second null hypothesis, providing further evidence about the performance of certain regression testing methodologies and techniques. Last, we detected that in most instances, technique effectiveness varies depending on test suite granularity.

More important from a practitioner's perspective, however, are implications for tradeoffs and factors involved when designing test suites and choosing granularities. The follow-

Techniques: Optimal and Random					
Variable: *APFD.*					
Source	SS	D.F.	MS	F	p
Granularity	33.19	3	11.06	0.42	0.74
Technique	4725.04	1	4725.04	179.10	0.00
Interaction	117.55	3	39.18	1.49	0.23
Error	1899.48	72	26.38		

Table 7: Prioritization Anova.

ing paragraphs address some of these implications, and help clarify the practical impact of the results (taking into consideration the threats to validity discussed in Section 3.4).

Reducing the test suite versus reducing overhead. Coarser granularity can greatly increase the efficiency of a test suite. For example, increasing granularity from G1 to G4 on the `emp-server` test suite saved an average of 270 minutes (73% time reduction). Finer granularity, however, is clearly more supportive of regression test selection and test suite reduction, since the effectiveness of these techniques diminishes as granularity increases. This tendency was evident for both programs (see Figure 1). For example, when the modified non-core entity RTS technique was applied to the G1 suite of `emp-server`, the suite's average execution time was reduced from 371 to 149 minutes (60% time reduction). When the same technique was applied to the G64 suite for `emp-server`, the average savings were less than 2%. Hence, finer granularity provides greater flexibility through larger numbers of small test cases that can be successfully manipulated by RTS and test suite reduction techniques to reduce the number of test cases to be executed.

Even when RTS and test suite reduction methodologies can save significant execution time, however, increasing test suite granularity by joining small test cases might be preferable to employing such methodologies. When test suite reduction was applied to the G1 test suite for `bash`, three out of five versions required less retesting time than their corresponding versions under G64 test suites. On the other hand, the savings obtained by applying test suite reduction to the `emp-server` G1 test suites were less than the savings generated by using level G64 suites, independent of version. These differences can be attributed to the amount of overhead in test suite execution required for each program. In our experiments, the savings generated by increases in granularity resulted primarily from reduction in the overhead associated with test setup and cleanup. (In other cases, another factor in overhead might be the cost of human intervention.) Test suites with larger granularity had fewer test cases, which reduced the overall overhead of the suite; this effect was more profound for `emp-server`, whose test cases carried more overhead than the `bash` test cases.

Note, however, the other side of the tradeoff: test suites with low overhead are not likely to yield time savings through increases in granularity. For such suites, potential savings through RTS or reduction may become the dominant factor in choosing granularity.

Diminishing returns of granularity increases. Even when finer granularity turns out to be better from an RTS or test suite reduction perspective, smaller test cases might not be as effective at detecting faults as larger ones. Larger test cases usually cover more code and, in our experiments, were more likely to execute faulty functions and to expose faults. Table 8 lists the total number of faults missed at each granularity for each program, and the average number of faults detected per test for retest-all. Clearly, fault detection effectiveness increases with granularity.

	Undetected Faults		*Avg. Faults Detected per Test*	
	emp-server	bash	emp-server	bash
G1	27	7	1.09	2.02
G4	12	3	1.16	2.08
G16	3	0	1.45	2.80
G64	0	0	2.89	4.52

Table 8: Fault Detection Effectiveness.

However, although increases in granularity provide greater fault detection effectiveness, there seems to be a point of diminishing returns at which granularity increments don't provide additional power. Furthermore, the effectiveness gains seem to become smaller in spite of our exponential granularity increments. In our experiments, `bash` test suites G16 and G64 detected the same numbers of faults, and only two versions of `emp-server` presented differences in fault detection between G16 and G64. The same argument applies to savings in execution time: there is a point of diminishing returns at which increasing granularity does not result in significant time savings.

Change characteristics and granularity. In our experiments we also discovered that fault location and likelihood of execution had an impact on the methodologies.

First, we found that the number of test cases through changed functions can greatly impact fault detection effectiveness. For example, on the `bash` G1 test suite, missed faults were located in functions executed by an average of 3% of the test cases, while exposed faults were located in functions executed by an average of 66% of the test cases. This difference can be overcome, however, if the test cases executing changed functions are effective at exposing faults. For example, on the `emp-server` G1 test suite, fewer than 2% of the test cases execute changed functions, but 26% of them expose faults.

Second, the percentage of changes located in core functionality impacts the effectiveness of RTS techniques. For example, for the G1 test suite on version 4 of `bash`, modified entity selection included all tests cases, while modified non-core entity selection included only 3% of the test cases.

Unresolved issues and opportunities. Several questions remain unanswered and new questions have emerged as a result of these experiments. First, we must be sensitive to the existence of metrics that capture other meaningful attributes impacted by test suite granularity. For example, test suites with finer granularity might facilitate fault localization. Our metrics do not reflect all possible impacts.

Second, we cannot fully explain the prioritization results and we realize that there are factors affecting variation in rate of fault detection that we are not capturing. Although we corroborated previous studies by providing additional empirical evidence about the potential of prioritization techniques in general, our expectation of greater APFD for finer granularities was true only for optimal prioritization.

Third, the greater fault detection effectiveness of coarser granularity test suites might be attributed (at least in part) to the execution of additional code which causes data state changes occurring in earlier stages of execution to be visible. It might be that smaller granularity test suites could be more effective if they were equipped with the right observers.

5. CONCLUSION

Writers of testing textbooks have long shown awareness that test suite granularity can affect the cost-effectiveness of testing. These effects can begin when testing the initial release of a system: success in finding faults in that release, as well as the amount of testing that can be accomplished, can vary based on test suite granularity. However, successful software evolves: the costs of testing that software are compounded over its lifecycle, and the opportunity to miss faults through inadequate regression testing occurs with each new release. It is therefore imperative to study the effects of test suite design across the entire software lifecycle.

Several test suite design factors, such as test suite size and adequacy criteria, have been empirically studied, but few have been studied with respect to evolving software. Several regression testing methodologies have been empirically studied, but few with respect to issues in test suite design. This paper brings the empirical study of test suite design and regression testing methodologies together, focusing on a particular design factor: test suite granularity. Our results highlight cost-benefits tradeoffs associated with granularity, and lay the groundwork for further empirical study.

We are continuing this family of experiments. We plan to obtain and create additional subject infrastructure, to experiment with wider samples of faulty versions, regression testing techniques, and test suite granularities, and to extend our measures to incorporate other cost-benefits factors. We also plan to consider the use of other groupings of test inputs. We then hope to use the data and results obtained to provide guidelines that will help practitioners design test suites that can be used more efficiently and effectively across the entire lifecycle of evolving systems.

ACKNOWLEDGEMENTS

This work was supported by the NSF Information Technology Research program under Awards CCR-0080898 and CCR-0080900 to University of Nebraska, Lincoln and Oregon State University, and by NSF Awards CCR-9703108 and CCR-9707792 to Oregon State University. We thank Xuemei Qiu, Satya Kanduri, and Srikanth Karre for helping prepare the `emp-server` and `bash` subjects.

REFERENCES

[1] J. Bach. Useful features of a test automation system (part iii). *Testing Techniques Newsletter*, Oct. 1996.

[2] B. Beizer. *Black-Box Testing*. John Wiley and Sons, New York, NY, 1995.

[3] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Reading, MA, 2000.

[4] T. Chen and M. Lau. Dividing strategies for the optimization of a test suite. *Info. Proc. Let.*, 60(3):135–141, Mar. 1996.

[5] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. 16th Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. Int'l. Symp. Softw. Testing and Analysis*, pages 102–112, Aug. 2000.

[7] S. Elbaum, J. Munson, and M. Harrison. CLIC: A tool for the measurement of software system dynamics. In *SETL Technical Report - TR-98-04.*, 04 1998.

[8] T. Graves, M. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. 20th Int'l. Conf. Softw. Eng.*, pages 188–197, Apr. 1998.

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

[10] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. and Meth.*, 2(3):270–285, July 1993.

[11] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In *Proc. Int'l. Symp. Softw. Testing and Analysis*, pages 135–145, Aug. 2000.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[13] C. Kaner, J. Falk, and H. Q. Nguyeen. *Testing Computer Software*. Wiley and Sons, New York, 1999.

[14] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proc. 22nd Int'l. Conf. Softw. Eng.*, pages 126–135, June 2000.

[15] E. Kit. *Software Testing in the Real World*. Addison-Wesley, Reading, MA, 1995.

[16] H. Leung and L. White. Insights into regression testing. In *Proc. Conf. Softw. Maint.*, pages 60–69, Oct. 1989.

[17] H. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proc. Conf. Softw. Maint.*, pages 290–300, Nov. 1990.

[18] D. Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., Sebastopol, CA, Nov. 1996.

[19] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley and Sons, New York, fourth edition, 1997.

[20] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. Twelfth Int'l. Conf. Testing Computer Softw.*, pages 111–123, June 1995.

[21] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. ACM*, 41(5):81–86, May 1988.

[22] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.

[23] C. Ramey and B. Fox. *Bash Reference Manual*. O'ReillyO'Reilly & Associates, Inc., Sebastopol, CA, 2.2 edition, 1998.

[24] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, Aug. 1996.

[25] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, Apr. 1997.

[26] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++ programs. *J. Softw. Testing, Verif., Rel.*, 10(2), June 2000.

[27] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. Int'l. Conf. Softw. Maint.*, pages 34–43, Nov. 1998.

[28] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, June 1998.

[29] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization. *IEEE Trans. Softw. Eng.*, Oct. 2001.

[30] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. Eighth Intl. Symp. Softw. Rel. Engr.*, pages 230–238, Nov. 1997.

[31] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Softw. Pract. and Exp.*, 28(4):347–369, Apr. 1998.

Appendix: Additional Anovas

Techniques: Modified non-core Entity and Modified Entity					
Variable: *Percentage of faults detected.*					
Source	SS	D.F.	MS	F	p
Granularity	15078.22	3	5026.07	19.92	0.00
Technique	90.31	1	90.31	0.36	0.55
Interaction	120.94	3	40.31	0.16	0.92
Error	18164.85	72	252.29		
Variable: *Percentage of time saved.*					
Source	SS	D.F.	MS	F	p
Granularity	7976.04	3	2658.68	8.24	0.00
Technique	11484.79	1	11484.79	35.61	0.00
Interaction	7934.77	3	2644.92	8.20	0.00
Error	23219.91	72	322.50		

Table 9: Selection Anovas.

Techniques: Retest-All and Modified Entity					
Variable: *Percentage of faults detected.*					
Source	SS	D.F.	MS	F	p
Granularity	12515.06	3	4171.69	15.23	0.00
Technique	0.00	1	0.00	0.00	1.00
Interaction	0.00	3	0.00	0.00	1.00
Error	19727.90	72	274.00		
Variable: *Percentage of time saved.*					
Source	SS	D.F.	MS	F	p
Granularity	0.08	3	0.03	0.64	0.59
Technique	0.15	1	0.15	3.83	0.05
Interaction	0.08	3	0.03	0.64	0.59
Error	2.82	72	0.04		

Table 10: Selection Anovas.

Techniques: Optimal and Additional Coverage					
Variable: *APFD.*					
Source	SS	D.F.	MS	F	p
Granularity	166.66	3	55.55	2.44	0.07
Technique	1475.35	1	1475.35	64.80	0.00
Interaction	60.38	3	20.13	0.88	0.45
Error	1639.22	72	22.77		

Table 11: Prioritization Anova.

Techniques: Random and Additional Coverage					
Variable: *APFD.*					
Source	SS	D.F.	MS	F	p
Granularity	149.72	3	49.91	1.02	0.39
Technique	919.82	1	919.82	18.76	0.00
Interaction	126.78	3	42.26	0.86	0.46
Error	3530.66	72	49.04		

Table 12: Prioritization Anova.