1 Introduction

Throughout their lifetimes, software systems undergo numerous changes. To make these changes, we have to understand these systems, identify and evaluate alternative modification strategies, implement the changes, and validate their correctness. In practice, the cost of these activities is enormous[1, 2, 5, 6].

Consider the problem of validating modified software. One common way to do this is to rerun tests from existing test suites (called regression testing). Although valuable, this is often very expensive. For instance, some companies must release software products for users who speak different languages. Typically, they release an initial version and then localize it for specific languages. Before releasing a localized version, they regression test it. Since localized versions must be available as soon as possible, regression testing time must be reduced. Another common practice is to run regression tests periodically (one company we know of regressions tests every third weekend for 24 to 36 hours per testing session). In this case, management might wish to test more frequently, but the time required is too great.

Because these problems are important and expensive, many researchers have investigated ways to solve them. One approach is to use program-analysis techniques, which analyze source code to help developers perform specific testing and maintenance tasks. Even though these techniques can partially or fully automate testing and maintenance, they're rarely used in practice. One reason for this is their current inability to scale to large systems.

Our preliminary research suggests that scalable, feasible, and cost-effective program-analysis-based testing and maintenance techniques can be developed. Of course, even if we can create techniques that appear to be scalable, they are not likely to be used without empirical evidence of their scalability and effectiveness. Moreover, our efforts to design such techniques are not likely to succeed without feedback from empirical studies.

Unfortunately, such large-scale empirical studies face serious obstacles. For instance, the tools being studied must process real source code, run efficiently, and handle the large volumes of data produced by program analyses. Few research prototypes meet these requirements.

In short, we believe that this problem will require the collaborative efforts of researchers in empirical studies, system development, and algorithm design. Therefore, we are conducting a long-term, collaborative research project, with the following objectives.

1. Construct a program-analysis infrastructure. We are constructing an extensible infrastructure for implementing and evaluating a wide variety of program-analysis-based testing and maintenance techniques. Because the infrastructure must support large-scale experimentation, we are collecting a repository of artifacts, including programs with multiple...
versions, test suites, test scripts, and fault data, that will serve as benchmark suites for use in experimentation.

2. **Develop scalable program-analysis techniques.** We are developing and evaluating several approaches to analyzing large systems, including system-level approaches that exploit demand-driven and layered analyses. We are also evaluating the trade-offs between storing intermediate program representations on secondary storage and recomputing this information.

3. **Perform large-scale experimentation.** We are conducting a family of experiments to compare the cost-benefits of existing approaches, evaluate the gains offered by our new approaches, and determine the value of experimental features of our infrastructure.

## 2 Plans for Large-Scale Experimentation

A distinguishing feature of our research approach is its emphasis on using experimentation to guide our research, not simply to demonstrate our tools. Most empirical studies in software engineering are used to demonstrate the feasibility or compare the performance of tools; rarely are they used to validate or quantify the fundamental assumptions that underlie the research. There are many experiments we could perform for this project; in this section we describe three specific families of experiments that we believe will answer important software engineering questions, and help guide our research.

### 2.1 Overall Experimental Strategy

Our experiments will have a common structure: we observe the application of several methods to a variety of test artifacts to perform one or more tasks. A particular method, set of test artifacts, or task is used as a control against which the others are compared. These methods, test artifacts, and tasks are the *independent* variables of our experiments. Other, *dependent* variables represent performance statistics that are gathered from every observation. These measures vary with the experiment and the hypotheses it is testing.

Our experimental designs attempt to avoid several common flaws: (1) naive experimental designs with poor statistical controls, (2) small sample sizes with only a single experimental run, and (3) lack of generalizability to industrial practice. We will take several steps to ensure the validity of our experiments: (1) we will use iterative experimental designs that adequately control many threats to the experiment’s internal and external validity; (2) we will conduct multiple runs of each experiment both to increase sample size and to understand the natural variance of the algorithms under study; (3) we will perform our experiments in multiple steps, starting with a simple experiment using smaller programs, followed by more experiments using larger programs, and finally by case studies using large commercial systems. The later phases of experimentation will be conducted as the infrastructure matures to support processing of larger programs.

### 2.2 Experiment 1. Comparing Regression Test Selection Algorithms

Two recent studies of the cost-effectiveness of regression test selection report contradictory results. Rothermel and Harrold conducted studies of their DejaVu tool for regression test selection [8]. They performed three studies in which they ran their tool on 132 versions of seven, 150 to 500 line C programs, on five versions of a 50,000 line C program, and on nine versions of a 2,000 line C program. Their results showed that their test selection algorithm could substantially reduce the size of a test suite (average reductions of 43%, 96%, and 67% in the three studies), and that the cost of selecting and then running the reduced test suite was far less than the cost of running the entire test suite.

Rosenblum and Weyuker, on the other hand, conducted a similar study using their TestTube system [7]. In that study, they examined 31 versions of the Korn Shell. One result was that
in 80% of the tests TestTube selected all the original tests. Furthermore, the cost of the TestTube analysis was two orders of magnitude greater than the cost saved by not running some of the tests.

We can make several observations about these studies. First, the analytical analyses provided with them could not have predicted the actual differences in their use; other types of analyses are needed. Second, some existing techniques may not be cost-effective, but we do not know which ones or under what conditions. Finally, because we do not have a good understanding of existing techniques, it is impossible to say whether new techniques in fact offer improvements. From these observations, we conclude that there is an overwhelming need for rigorous empirical studies to help determine the fundamental factors determining the costs and relative benefits of these techniques.

Two widely studied classes of regression test selection methods are safe methods and coverage-based methods. Safe methods guarantee that all fault-revealing test cases are selected. Coverage-based methods attempt to find test cases that meet some coverage criterion. Alternatively, test selection might be avoided altogether and testing performed with all tests: this is the retest-all approach that is currently prevalent in practice. In this experiment, we explore the cost-benefit tradeoffs among these three classes of approaches.

One hypothesis underlying this experiment is that, in many cases, the savings provided by test selection are outweighed by the cost of performing the necessary analyses. Another hypothesis is that coverage-based test selection results in smaller test sets than safe test selection, but sacrifices considerable fault-detection ability.

The task for this experiment is regression test selection using either a Random(N)$^1$, safe, retest-all, or coverage-based method. Initially, we will use the DejaVu system for the safe method and edge coverage as the coverage-based method. For each test run, we will measure (1) the percentage of tests in the original test set that reveal a fault; (2) the cost of performing all tests in the original test suite; (3) the proportion of tests selected by the test selection method; (4) the proportion of fault-revealing tests that were selected to fault-revealing tests in the original test suite (inclusiveness); (5) the proportion of fault-revealing tests that were selected to total tests that were selected (precision); and (6) the cost of performing all tests in the original test suite minus the cost of selecting tests and then running them (test selection savings). This experiment uses a full factorial design. With this design, each test selection method is applied to each program and its test suite.

Each experimental run consists of a control run and an experimental run. In the control run, the entire test suite is rerun, and performance statistics are calculated. In the experimental run, one of the test selection methods is used. The selected tests are then run and performance statistics are calculated.

Two sets of data are important to our study: the cost and the fault-detection summaries. The cost summary captures the time required to select and run a subset of the original test suite. Using this information we can calculate the savings offered by a particular test selection method. The fault-detection summaries tell us which test cases in the original test suite are fault-revealing and which of these tests were selected by the test selection methods. We use this information to calculate the fault-detection ability of each test selection method. (For safe methods, all fault-revealing tests are guaranteed to be selected if certain assumptions are met. These experiments will provide insight into whether these assumptions are met in practice.)

2.3 Experiment 2. Demand-Driven Approaches to Large-Scale Program Analysis

We suspect that in many cases the analysis of entire programs is wasted because only a small portion of the analysis is actually needed to complete the task. Previous research by us and others suggests that demand-driven approaches can be used to build only those portions of graphs that are actually needed. Before exploring this strategy much further, we want to know whether typical analysis tasks can be substantially improved using demand-driven strategies. We also want to know whether these strategies produce savings that increase as the system grows; this

$^1$Each test in the test suite is selected with probability N, where N is either 0.25, 0.5, or 0.75.
would imply that such approaches are more scalable than exhaustive approaches. If either of these conjectures is false, then we will concentrate on other approaches to scalability.

In this experiment, we perform two tasks. One task involves impact analysis. That is, given a proposed change, determine which statements will be affected. The second task is safe regression test selection. The experiment investigates impact analysis and regression test selection using demand-driven and exhaustive analysis approaches. We will investigate two types of demand-driven approaches: database queries with magic-set optimizations, and a standard procedural approach. For each test run we will measure (1) the size of the complete analysis graph; and (2) the proportion of the graph constructed during the analysis.

The data set that is most important to us in this experiment is the analysis cost summaries. The cost summary captures the size of the graph needed to complete the experimental task, and the size of the graph generated by the demand-driven approach in order to complete the same task.

2.4 Experiment 3. Reuse-Driven Approaches to Large-Scale Program Analysis

Another possible approach for scaling analysis processes is to reuse the results of previous analyses. Harrold and Rothermel [3] showed one way to exploit reuse in an algorithm for resolving aliases. We hypothesize that, in practice, there are enough opportunities to reuse previous analyses that storing and retrieving the extra information needed during the integration phase is justified. We also hypothesize that these benefits increase with the complexity of the analysis task and with the size of the system. If these hypotheses are correct, this implies that reuse may be a scalable approach, but also that for some tasks, recomputation rather than storage is most cost-effective.

This experiment involves two analysis tasks — calculating data-dependence graphs and calculating control-dependence graphs — using either a reuse-driven or an exhaustive approach. The experiment also involves two maintenance tasks that use these analysis tasks: impact analysis for regression test selection, and slicing for debugging. For each maintenance task, we investigate the use of each analysis task; we measure (1) the size of the complete analysis graph; (2) the proportion of the graph reused during the analysis; and (3) the system time needed to complete the analysis.

We will compile two data sets for this study. One cost summary will reflect the size of the complete analysis graphs and the proportion of it that was reused rather than regenerated. The other cost summary will capture system times needed to perform the analyses.

3 Summary

We expect several tangible results from this work. First, if our research shows that scalable program-analysis-based techniques are possible, then we will provide some. If our research does not support this belief, we will be able to suggest why it did not, how further work might, or why it isn’t possible. In addition, as this work proceeds, we will be able to provide significant empirical data on the cost-benefits of specific techniques, such as selective regression testing.

Second, we will develop an extensible infrastructure for use in experimentation with these techniques. We will provide the system and a data repository with which others can experiment.

Finally, our discoveries will be a starting point for future work on program-analysis-based testing and maintenance. Even if we discover that program analysis techniques do not yield scalable tools for testing and maintenance of whole programs, they may have applications to smaller, but still important, software systems. For example, they may apply to component-based software systems: systems composed primarily of modules that encapsulate both data and functionality and are configurable through parameters at run-time [4]. Like other software these components require testing and maintenance; moreover, systems built from components must be tested and maintained.
4 Acknowledgements

This work was supported in part by a grant from Microsoft, Inc., by NSF under NYI award CCR-9696157 to Ohio State University, and by an Ohio State University Research Foundation Seed Grant. It was also supported in part by by NSF Grant SBR-9529926 to the National Institute of Statistical Sciences, and by National Science Foundation Faculty Early Career Development Award, CCR-9501354.

References


