Experience With Regression Test Selection

Gregg Rothermel
Department of Computer Science
Oregon State University
Dearborn Hall 307-A
Corvallis, OR 97331
grother@cs.orst.edu

Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
395 Dreese Lab, 2015 Neil Avenue
Columbus, OH 43210
harrold@cis.ohio-state.edu

1 Introduction

Regression testing is a maintenance process that attempts to validate modified software, and ensure that no new errors are introduced into previously tested code. Regression testing is expensive; it can account for as much as one-half of the cost of software maintenance [19]. One approach to regression testing is selective retest, which addresses two problems: (1) the problem of selecting tests from an existing test suite, and (2) the problem of determining where additional tests may be required. We have developed a new selective retest technique that addresses the first problem [26]. Our algorithms construct control flow graphs for a procedure or program and its modified version, and use these graphs to select tests from the original test suite that execute changed code.

To investigate the application of our technique in practice, we implemented one of our algorithms as a tool called DejaVu. We conducted empirical studies with our technique, by applying DejaVu to various programs, modified versions, and test suites. Our empirical results support several conclusions about regression test selection. In particular, our results suggest that our technique can significantly reduce the cost of regression testing a modified program.

2 Background

Research on regression testing spans a wide variety of topics, including test environments and automation [6, 8, 15, 33], capture-playback mechanisms [22], test suite management [11, 22, 29], code size reduction [4], and regression testability [19].

Most recent research on regression testing, however, concerns selective retest techniques. Selective retest techniques reduce the cost of regression testing by selectively reusing tests from a program’s existing test suite. Selective retest techniques differ from the retest-all technique, which runs all tests in the existing test suite. Leung and White [21] show that a selective retest technique is more economical than the retest-all technique only if the cost of selecting a reduced subset of tests is less than the cost of running the tests that the selective retest technique omits. Numerous selective retest techniques have been proposed [1, 2, 3, 5, 7, 9, 10, 13, 14, 17, 18, 20, 23, 28, 29, 30, 32]. These techniques have been analytically compared and evaluated in [27]; however, to date, only three of these techniques [24, 28, 31] have been subjected to empirical validation.

We have developed a family of regression test selection algorithms. Our most basic algorithm builds control flow graphs\(^1\) for a program \(P\) and modified version \(P'\), collects test traces that associate tests in \(T\) with edges in the graph for \(P\), and performs synchronous depth-first traversals of the two graphs, comparing the program statements associated with nodes that are simultaneously reached in the two graphs. When the algorithm discovers a pair of nodes \(N\) and \(N'\) in the graphs for \(P\) and \(P'\), respectively, such that the statements associated with \(N\) and \(N'\) are not lexically identical, the algorithm selects all tests from \(T\) that, in \(P\), reached \(N\). This approach identifies tests that reach code that is new in, or modified for, \(P'\), and tests that formerly reached code that has been deleted from \(P\). A significant feature of our approach is that, un-

\(^1\) A control flow graph is a directed graph in which nodes represent program statements, and edges represent the flow of control between statements.
der certain well-defined conditions, our algorithms are safe: they select every test from the original test suite that can expose faults in the modified program. In [25, 26], we describe our algorithms in detail, and present proofs of their correctness and safety. In [27], we show analytically that among existing regression test selection algorithms, only three [7, 14, 17] can possibly be safe. These approaches each depend for their safety upon the same conditions on which our approach depends. Moreover, among safe approaches, our approach is the most precise: it selects the fewest unnecessary tests.

To investigate the use of our technique in practice and to facilitate experimentation, we implemented our basic algorithm as a tool called DejaVu. To obtain the program analysis information required for this implementation, we used a program analysis system called Aristotle [12]. Aristotle provides several varieties of program analysis tools, code instrumenters, and graphical representations of programs. To manage test suites and automate test execution, we implemented a test database management system and capture playback tools. Given this external support, DejaVu was easy to implement, requiring only 1220 lines of C code. We describe this implementation in detail in [25].

3 Empirical Studies

We wished to investigate the application of our algorithms in practice. In particular, we wished to determine the size of the test sets selected by our algorithms, the cost of running the algorithms, the savings (if any) that can be achieved using the algorithms, and the parameters that govern the effectiveness and efficiency of the algorithms. To address these issues, we performed three empirical studies.

Study 1: Regression Test Selection

Our first study investigated the efficacy of DejaVu on a set of small but nontrivial programs. Hutchins et al. [16] report the results of an experiment on the effectiveness of dataflow- and controlflow-based test adequacy criteria in which they used seven C programs, 132 versions of these programs, and test pools that they created for each program; Table 1 gives information about these experimental subjects. Because Hutchins et al. investigated error detection capability, their study used faulty versions of base programs; we think of these versions as ill-fated attempts to modify the base programs. These subject programs presented some disadvantages for our study because they employ only erroneous modifications, use constructed faults rather than real faults, and use only faults that yield meaningful detection rates. However, the subjects also had advantages for an initial study: namely, we could easily obtain them and use our tools on them, and the seeded faults do model real faults.

For each base program, we ran all tests in the test suite for the program on an instrumented version of the code, and saved the output and test traces. We also calculated the time required to construct control flow graphs for the programs and modified versions. We ran DejaVu on each base program, paired with each version, in turn, with the test as input, timing the runs and redirecting output to a results file. We then ran all selected tests on the modified versions, ran all unselected tests on the modified versions, and ran all tests on the modified versions. We counted the number of tests that actually revealed faults in modified versions and collected timings for test runs. We repeated each experiment five times for each (base program, modified program) pair, and averaged our results over these runs; all timings that we report for this study use these average results.

The graph on the left of Figure 1, which depicts test selection statistics for DejaVu, shows that DejaVu reduced test set size; on average, 44.4% of the tests were eliminated. The graph on the right of the figure, which depicts timing statistics for DejaVu, shows that this reduction in test set size resulted in a reduction in the time required to run tests; on average over all programs and modified versions, testing time was reduced from 9 minutes and 46 seconds to 5 minutes and 45 seconds.

We observed a wide range of variance in test selection results for trials of DejaVu on individual programs and modified versions. For example, program replace has a test pool that contains 5542 tests, of which DejaVu selects, on average, 2399 (43.29%). However, the selected sets ranged in size from 52 to 5542, with no size range predominant. In contrast, sched2 has a test pool that contains 2680 tests, of which DejaVu selects, on average, 2508 (93.58%). On eight of the ten modified versions of sched2, DejaVu selects at least 90% of the existing tests. By examining the subject programs and test suites, we conclude that the variance in the size of the test sets selected by DejaVu is affected by (1) the struc-
Table 1: Subject programs and statistics for modified versions for Study 1.

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Number of Procedures</th>
<th>Lines of Code</th>
<th>Number of Versions</th>
<th>Number of Tests</th>
<th>Description of Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>21</td>
<td>512</td>
<td>32</td>
<td>5542</td>
<td>pattern replacement</td>
</tr>
<tr>
<td>usl.123</td>
<td>20</td>
<td>472</td>
<td>7</td>
<td>4056</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>usl.128</td>
<td>21</td>
<td>399</td>
<td>10</td>
<td>4071</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>sched2</td>
<td>16</td>
<td>301</td>
<td>10</td>
<td>2680</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>sched1</td>
<td>18</td>
<td>292</td>
<td>9</td>
<td>2650</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>totinfo</td>
<td>16</td>
<td>440</td>
<td>23</td>
<td>1054</td>
<td>information measure</td>
</tr>
<tr>
<td>tcas</td>
<td>8</td>
<td>141</td>
<td>41</td>
<td>1578</td>
<td>altitude separation</td>
</tr>
</tbody>
</table>

Figure 1: Test selection statistics (left) and timing statistics (right) for Study 1.

ture of the original program, (2) the location of the modifications, and (3) the extent of the code coverage achieved by tests in the test suite.

Because the modified versions used in this study contained erroneous modifications, we were able to make some observations about the fault-revealing capabilities of regression test selection techniques. For the faulty modified versions of this study, a very small percentage of the tests for the programs reveal the faults. For example, consider program replace, with a test suite of 5542 tests. For version 26 of replace, 302 of the 5542 tests reveal the fault in that version. DejaVu finds that 1012 of the 5542 tests of the version execute changed code and selects them. A test selection method that selects, from the set of tests that reach changed code, only one test, has only a 29.8% chance of selecting one of the 302 fault-revealing tests. Next, consider version 19 of replace. Although 4658 of the 5542 tests of replace execute changed code in the version, only 3 of these tests reveal the fault in that version. A test selection method that selects only one of the 4658 tests that exercise the changed code in the version has only a .064% chance of selecting a test that exposes the fault. In either case, DejaVu necessarily exposes the fault. Study 1 exhibited many other comparable cases.

Study 2: Regression Test Selection on a Larger Scale

Our second study investigated the efficacy of our test selection algorithms on a larger software system, player – a component of the software distribution for the internet-based game, Empire. Empire consists of several separate executables; the player program, which processes transactions, is the largest of these. Since its creation in 1986 as an internet-based game, Empire has been rewritten many times. A recent major revision was released in the early 1990s. Since this release, several modified versions of that revision have been created, to fix bugs or add functionality. Most of these versions alter the player program. Table 2 summarizes significant statistics about player and Table 3 gives...
details on five versions of player that we used for our study.

To construct a test suite for player, we used the Empire information files, which describe the 154 commands that are recognized by the player executable, and discuss parameters and special side effects of each command. We treated the information files as informal specifications; for each command, we used its information file to construct versions of the command that exercise all parameters and special features, and test erroneous parameter values and conditions. Because the complexity of commands and parameters varies widely over the set of player commands, this process yielded between 1 and 30 tests of each command, and ultimately produced a test suite of 1035 functional tests. To avoid the possibility of biasing the test suite, we constructed the suite prior to examining the code of the modified versions.

Our experimental procedure for Study 2 was similar to that of Study 1, except that we had to simulate the actions of DejaVu for 15% of the player functions due to limitations in our prototype. Thus, although the numbers of tests selected for the subject programs are exact, and the times required to run tests are exact, we obtained times for program analysis and test selection by computing results for 85% of the code, and multiplying the results by 1.17.

The graph on the left of Figure 2, which depicts the test selection statistics for DejaVu for Study 2, shows that DejaVu averaged a savings in test size of over 95%. By our approximation, CFG construction for both base and modified versions of player requires 23 minutes and 17 seconds. We also estimated the time required by DejaVu (independent of graph construction costs), by running DejaVu on the player files that our prototype processes, and scaling the results. If the results scale linearly, we calculate an upper bound of 1 minute and 28 seconds to run our test selection algorithms on player.
Using our estimates, the graph on the right in Figure 2 projects the savings that our test selection algorithms provide for \texttt{player} and its modified versions. As the graph shows, assuming that the analysis of \texttt{player} requires the 24 minutes and 45 seconds we estimated, our algorithms, in every case, produce at least an 82\% savings in regression testing time. However, in this study, our measurements of test execution time did not account for the time that a tester would spend validating test results; our algorithms would thus provide greater savings in practice. Much of the time saved is time that some human need not spend performing unnecessary testing.

**Study 3: Regression Test Selection on Commercial Software**

Our third study investigated the efficacy of \texttt{DejaVu} on a small commercial program. We obtained the program, nine modified versions of the program, and the test suite that had been used to regression test the program. Table 4 describes the subject program, and Table 5 shows statistics for modified versions.

The commercial program presented some advantages over those used in Studies 1 and 2 because, although the program is relatively small, it is a program used in a commercial software system, and its versions and test suites were not constructed for the purpose of the study. However, because the software contains C constructs that our prototype implementation does not handle, we could only simulate the actions of \texttt{DejaVu} on the subject program and modified versions. The graph in Figure 3, which depicts the test selection statistics for our algorithm in this study, shows that in this case, \texttt{DejaVu} averaged a savings in test set size of about 67\%. Because we simulated the actions of \texttt{DejaVu} for this study, we could not gather timing statistics. However, we believe that using \texttt{DejaVu} will result in an overall reduction in the time required to test the modified versions of the program.

An interesting additional observation is that for two of the nine modified versions considered in the study (versions 2 and 5), \texttt{DejaVu} selected no tests to rerun. The fact that \texttt{DejaVu} selected no tests for these versions means that, in these cases, no test in the test suite actually executed code modified for that version. Thus, even if a retest-all approach were used on these programs, crucial statements in the modified versions would not be tested.

**4 Conclusions**

Our empirical studies support several conclusions:

- Regression test selection in general, and safe regression test selection in particular, can be efficient and effective in practice. Our algorithms can significantly reduce the time required to regression test modified software, even when we consider the cost of the analysis performed to select tests.

- Regression test selection algorithms can yield even greater savings when applied to large, complex programs than when applied to small, simple programs.

- There exist programs, modified versions, and test suites for which test selection offers little in the way of savings. The factors that affect the effectiveness of test selection techniques include the structure of programs, the nature of the modifications made to those programs, and the type of coverage attained by tests.

- Selecting tests from an existing test suite is not necessarily sufficient for adequate regression testing; test selection alone may leave modified code untested.

Our results support the following additional hypotheses about regression test selection:

- Test selection at the system test level provides greater opportunities for savings than test selection at the unit test level; in fact, unit level test selection may not offer savings sufficient to justify its cost. Although we have not described all of our data here, additional studies on unit level test selection performed in conjunction with Study 1 support this hypothesis. If true, this result is important, because many existing selective retest techniques are strictly unit level [27].

- Regression test selection techniques that select minimal test sets through modified code may be significantly less effective than safe techniques at revealing faults.
<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Number of Functions</th>
<th>Lines of Code</th>
<th>Number of Versions</th>
<th>Number of Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>commercial</td>
<td>27</td>
<td>2145</td>
<td>9</td>
<td>388</td>
</tr>
</tbody>
</table>

Table 4: Subject program for Study 3.

<table>
<thead>
<tr>
<th>Version Number</th>
<th>Functions Modified</th>
<th>Lines of Code Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>264</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>245</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 5: Modified versions for Study 3.

Figure 3: Test selection statistics for Study 3.

The work we have described here raises several additional questions. First, because there are cases in which test selection does offer little savings, we might question the wisdom of performing test selection analysis on programs simply to discover that test selection cannot aid in their retesting; such analysis may not, in that case, involve time well spent. This suggests the merit of finding plausible predictors of test selection effectiveness. Such predictors must be able to determine whether the benefits of selection will outweigh its costs, but they should also account for the cost of omitting tests that may reveal faults – especially because safe techniques do not omit such tests.

Second, although our second study shows that test selection can be effective even for moderately large software systems, we have not yet experimented with truly large systems, nor have we experimented with a sufficiently large base of programs, modified programs, or test suites.

Third, a question that is often raised is whether, given adequate automation of the testing process, test selection is necessary at all. Our studies illustrate cases for which test suites require (at most) six hours to complete. When such test suites are fully automated, saving execution time for the suites may not be too important. However, when test suites require hours, days, or weeks of manual labor, even small savings in regression testing effort may be worthwhile. Even when test execution is fully automated, the desire to complete the test execution more quickly (e.g., overnight rather than in
20 hours) can justify the use of test selection.

Fourth, our work considers only the problem of code-based test selection; we believe that specification-based selection is also important.

Fifth, experimentation comparing various test selection techniques would be useful. This experimentation should also investigate the fault-revealing capabilities of test selection techniques.

For the sake of fairness, we mention the following limitations on our studies. We have considered only a small sample of the universe of possible programs, modified programs, and test suites. We make no claim that this sample is normally distributed. Study 1 used constructed versions and tests. The test suites used in Study 1 were unrealistically large, because in fact we used the “test pools” provided by Siemens, whereas the Siemens researchers, in their studies, had used these pools as a population from which to select a large number of smaller test suites. In Study 2, where our goal was to investigate whether there were larger programs for which our technique would be effective, we deliberately chose a subject for which we believed our algorithms would perform well. However, this subject was a real program, and is representative of a significant class of real programs. In this study, we constructed a test suite; however, this test suite is realistic; it represents the sort of functional test suite we would use in practice to test the program. Due to limitations in our prototype, our second study required some estimation. However, we were careful to estimate fairly and conservatively, so that our results, if erroneous, understate the effectiveness and efficiency of our technique, rather than overstating it.

5 Acknowledgements

This work was partially supported by NSF under grant CCR-9357811 to The Ohio State University.

References


