Module 2: Reading assignment (INL4.4)

This document provides the instructions for the Module 2 reading assignment (INL4.4) in DVA434.

Once you have completed the assignment in the form of a presentation, please email to wasif.afzal@mdh.se. The deadline for submission is 05-11-2015.

Please note that you will be presenting this presentation as part of INL3 (campus day).

Instructions:

1. In this assignment, you will read three papers. These papers are as following (These papers are attached below with these instructions):
    a. L. Inozemtseva and R. Holmes. *Coverage is not strongly correlated with test suite effectiveness*. In Proceedings of the 36th International Conference on Software Engineering (ICSE'14), 2014.
    b. C. Apa, O. Dieste, E. G. Espinosa G. and E. R. Fonseca C. *Effectiveness for detecting faults within and outside the scope of testing techniques: an independent replication*. Journal of Empirical Software Engineering, Vol. 19, No. 2, 2014.
    c. J. Offutt and C. Alluri. *An industrial study of applying input space partitioning to test financial calculation engines*. Journal of Empirical Software Engineering, Vol. 19, No. 3, 2014.
2. Please prepare a presentation (maximum of 15 slides) where you summarize and reflect on these papers. Think about answering the following aspects:
    a. What are the goals/objectives/research questions addressed by each paper?
    b. Do you, being an industrial professional, agree with these goals/objectives/research questions as being important to investigate from a practical point of view? If yes, why and if no, why not?
    c. What is the research method used in these papers and do you agree with how authors go about executing their study designs? From an industrial perspective, do their study designs reflect what typically happens in real world software testing?
    d. Do you agree with how test effectiveness and efficiency is measured in these papers? Do you see a possibility of using these measures in your profession or if they would complement to those already in practice?
    e. What are the outcomes/results of these papers? Are you surprised with the results? Can you criticize the results as not being representative of what happens in real world software testing?

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

We have extended these studies by evaluating the relationship between test suite size, coverage, and effectiveness for large Java programs. Our study is the largest to date in the literature: we generated 31,000 test suites for five systems consisting of up to 724,000 lines of source code. We measured the statement coverage, decision coverage, and modified condition coverage of these suites and used mutation testing to evaluate their fault detection effectiveness.

We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics—*product metrics*

General Terms

Measurement

Keywords

Coverage, test suite effectiveness, test suite quality

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
http://dx.doi.org/10.1145/2568225.2568271

1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Unfortunately, these studies do not agree on the strength of the relationship between these test suite characteristics. In addition, three issues with the studies make it difficult to generalize their results. First, some of the studies did not control for the size of the suite. Since coverage is increased by adding code to existing test cases or by adding new test cases to the suite, the coverage of a test suite is correlated with its size. It is therefore not clear that coverage is related to effectiveness independently of the number of test cases in the suite. Second, all but one of the studies used small or synthetic programs, making it unclear that their results hold for the large programs typical of industry. Third, many of the studies only compared adequate suites; that is, suites that fully satisfied a particular coverage criterion. Since adequate test suites are rare in practice, the results of these studies may not generalize to more realistic test suites.

This paper presents a new study of the relationship between test suite size, coverage and effectiveness. We answer the following research questions for large Java programs:

Research Question 1. *Is the effectiveness of a test suite correlated with the number of test cases in the suite?*

Research Question 2. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is ignored?*

Research Question 3. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is held constant?*

The paper makes the following contributions:

- A comprehensive survey of previous studies that investigated the relationship between coverage and effectiveness (Section 2 and accompanying online material).

Table 1: Summary of the findings from previous studies.

Citation	Languages	Largest Program	Coverage Types	Findings
[15, 16]	Pascal	78 SLOC	All-use, decision	All-use related to effectiveness independently of size; decision is not; relationship is highly non-linear
[17]	Fortran Pascal	78 SLOC	All-use, mutation	Effectiveness improves with coverage but not until coverage reaches 80%; even then increase is small
[14]	C	5,905 SLOC	All-use, decision	Effectiveness is correlated with both all-use and decision coverage; increase is small until high levels of coverage are reached
[39]	C	<2,310 SLOC	Block	Effectiveness is more highly correlated with block coverage than with size
[24]	C	512 SLOC	All-use, decision	Effectiveness is correlated with both all-use and decision coverage; effectiveness increases more rapidly at high levels of coverage
[6]	C	4,512 SLOC	Block, c-use, decision, p-use	Effectiveness is moderately correlated with all four coverage types; magnitude of the correlation depends on the nature of the tests
[3]	C	5,000 SLOC	Block, c-use, decision, p-use	Effectiveness is correlated with all four coverage types; effectiveness rises steadily with coverage
[31]	C C++	5,680 SLOC	Block, c-use, decision, p-use	Effectiveness is correlated with all four coverage types but the correlations are not always strong
[19, 37]	C Java	72,490 SLOC	AIMP, DBB, decision, IMP, PCC, statement	Effectiveness correlated with coverage; effectiveness correlated with size for large projects
[5]	C	4,000 SLOC	Block, c-use, decision, p-use	None of the four coverage types are related to effectiveness independently of size
[20]	Java	$O(100,000)$ SLOC	Block, decision, path, statement	Effectiveness correlated with coverage across many projects; influence of project size unclear

- Empirical evidence demonstrating that there is a low to moderate correlation between coverage and effectiveness when suite size is controlled for and that the type of coverage used has little effect on the strength of the relationship (Section 4).
- A discussion of the implications of these results for developers, researchers and standards bodies (Section 5).

2. RELATED WORK

Most of the previous studies that investigated the link between test suite coverage and test suite effectiveness used the following general procedure:

1. Created faulty versions of one or more programs by manually seeding faults, reintroducing previously fixed faults, or using a mutation tool.
2. Created a large number of test suites by selecting from a pool of available test cases, either randomly or according to some algorithm, until the suite reached either a pre-specified size or a pre-specified coverage level.
3. Measured the coverage of each suite in one or more ways, if suite size was fixed; measured the suite's size if its coverage was fixed.
4. Determined the effectiveness of each suite as the fraction of faulty versions of the program that were detected by the suite.

Table 1 summarizes twelve studies that considered the relationship between the coverage and the effectiveness of a test suite, ten of which used the general procedure just described. Eight of them found that at least one type of coverage has some correlation with effectiveness independently of size; however, not all studies found a strong correlation, and most found that the relationship was highly non-linear. In addition, some found that the relationship only appeared at very high levels of coverage. For brevity, the older studies from Table 1 are described more fully in accompanying materials[1]. In the remainder of this section, we discuss the three most recent studies.

At the time of writing, no other study considered any subject program larger than 5,905 SLOC[2]. However, a recent study by Gligoric et al. [19] and a subsequent master's thesis [37] partially addressed this issue by studying two large Java programs (JFreeChart and Joda Time) and two large C programs (SQLITE and YAFFS2) in addition to a number of small programs. The authors created test suites by sampling from the pool of test cases for each program. For the large programs, these test cases were manually written by developers; for the small programs, these test cases were automatically generated using various tools. Suites were created

[1] `http://linozemtseva.com/research/2014/icse/coverage/`

[2] In this paper, source lines of code (SLOC) refers to executable lines of code, while lines of code (LOC) includes whitespace and comments.

in two ways. First, the authors specified a coverage level and selected tests until it was met; next, the authors specified a suite size and selected tests until it was met. They measured a number of coverage types: statement coverage, decision coverage, and more exotic measurements based on equivalent classes of covered statements (dynamic basic block coverage), program paths (intra-method and acyclic intra-method path coverage), and predicate states (predicate complete coverage). They evaluated the effectiveness of each suite using mutation testing. They found that the Kendall τ correlation (see Section 4.2) between coverage and mutation score ranged from 0.452 to 0.757 for the various coverage types and suite types when the size of the suite was not considered. When they tried to predict the mutation score using suite size alone, they found high correlations (between 0.585 and 0.958) for the four large programs with manually written test suites but fairly low correlations for the small programs with artificially generated test suites. This suggests that the correlation between coverage and effectiveness in real systems is largely due to the correlation between coverage and size; it also suggests that results from automatically generated and manually generated suites do not generalize to each other.

A study by Gopinath et al. [20] accepted to the same conference as the current paper did not use the aforementioned general procedure. The authors instead measured coverage and test suite effectiveness for a large number of open-source Java programs and computed a correlation across all programs. Specifically, they measured statement, block, decision and path coverage and used mutation testing to measure effectiveness. The authors measured these values for approximately 200 developer-generated test suites – the number varies by measurement – then generated a suite for each project with the Randoop tool [36] and repeated the measurements. The authors found that coverage is correlated with effectiveness across projects for all coverage types and for both developer-generated and automatically-generated suites, though the correlation was stronger for developer-written suites. The authors found that including test suite size in their regression model did not improve the results; however, since coverage was already included in the model, it is not clear whether this is an accurate finding or a result of multicollinearity[3].

As the above discussion shows, it is still not clear how test suite size, coverage and effectiveness are related. Most studies conclude that effectiveness is related to coverage, but there is little agreement about the strength and nature of the relationship.

3. METHODOLOGY

To answer our research questions, we followed the general procedure outlined in Section 2. This required us to select:

1. A set of subject programs (Section 3.2);
2. A method of generating faulty versions of the programs (Section 3.3);
3. A method of creating test suites (Section 3.4);
4. Coverage metrics (Section 3.5); and
5. An effectiveness metric (Section 3.6).

We then measured the coverage and effectiveness of the suites to evaluate the relationship between these characteristics.

[3]The amount of variation 'explained' by a variable will be less if it is correlated with a variable already included in the model than it would be otherwise.

3.1 Terminology

Before describing the methodology in detail, we precisely define three terms that will be used throughout the paper.

- **Test case:** one test in a suite of tests. A test case executes as a unit; it is either executed or not executed. In the JUnit testing framework, each method that starts with the word `test` (JUnit 3) or that is annotated with `@Test` (JUnit 4) is a test case. For this reason, we will use the terms *test method* and *test case* interchangeably.
- **Test suite:** a collection of test cases.
- **Master suite:** the whole test suite that was written by the developers of a subject program. For example, the master suite for Apache POI contains 1,415 test cases (test methods). The test suites that we create and evaluate are strict subsets of the master suite.

3.2 Subject Programs

We selected five subjects from a variety of application domains. The first, Apache POI [4], is an open source API for manipulating Microsoft documents. The second, Closure Compiler [7], is an open source JavaScript optimizing compiler. The third, HSQLDB [23], is an open source relational database management system. The fourth, JFreeChart [25], is an open source library for producing charts. The fifth, Joda Time [26], is an open source replacement for the Java `Date` and `Time` classes.

We used a number of criteria to select these projects. First, to help ensure the novelty and generalizability of our study, we required that the projects be reasonably large (on the order of 100,000 SLOC), written in Java, and actively developed. We also required that the projects have a fairly large number of test methods (on the order of 1,000) so that we would be able to generate reasonably sized random test suites. Finally, we required that the projects use Ant as a build system and JUnit as a test harness, allowing us to automate data collection.

The salient characteristics of our programs are summarized in Table 2. Program size was measured with `SLOCCount` [38]. Rows seven through ten provide information related to mutation testing and will be explained in Section 3.3.

3.3 Generating Faulty Programs

We used the open source tool PIT [35] to generate faulty versions of our programs. To describe PIT's operation, we must first give a brief description of mutation testing.

A **mutant** is a new version of a program that is created by making a small syntactic change to the original program. For example, a mutant could be created by modifying a constant, negating a branch condition, or removing a method call. The resulting mutant may produce the same output as the original program, in which case it is called an **equivalent mutant**. For example, if the equality test in the code snippet in Figure 1 were changed to `if (index >= 10)`, the new program would be an equivalent mutant.

Mutation testing tools such as PIT generate a large number of mutants and run the program's test suite on each one. If the test suite fails when it is run on a given mutant, we say that the suite **kills** that mutant. A test suite's **mutant coverage** is then the fraction of non-equivalent mutants that it kills. Equivalent mutants are excluded because they cannot, by definition, be detected by a unit test.

If a mutant is not killed by a test suite, manual inspec-

Table 2: Salient characteristics of our subject programs.

Property	Apache POI	Closure	HSQLDB	JFreeChart	Joda Time
Total Java SLOC	283,845	724,089	178,018	125,659	80,462
Test SLOC	68,932	93,528	18,425	44,297	51,444
Number of test methods	1,415	7,947	628	1,764	3,857
Statement coverage (%)	67	76	27	54	91
Decision coverage (%)	60	77	17	45	82
MC coverage (%)	49	67	9	27	70
Number of mutants	27,565	30,779	50,302	29,699	9,552
Number of detected mutants	17,935	27,325	50,125	23,585	8,483
Number of equivalent mutants	9,630	3,454	177	6,114	1,069
Equivalent mutants (%)	35	11	0.4	21	11

```
int index = 0;
while (true) {
    index++;
    if (index == 10) {
        break;
    }
}
```

Figure 1: An example of how an equivalent mutant can be generated. Changing the operator == to >= will result in a mutant that cannot be detected by an automated test case.

tion is required to determine if it is equivalent or if it was simply missed by the suite[4]. This is a time-consuming and error-prone process, so studies that compare subsets of a test suite to the master suite often use a different approach: they assume that any mutant that cannot be detected by the master suite is equivalent. While this technique tends to overestimate the number of equivalent mutants, it is commonly applied because it allows the study of much larger programs.

Although the mutants generated by PIT simulate real faults, it is not self-evident that a suite's ability to kill mutants is a valid measurement of its ability to detect real faults. However, several previous and current studies support the use of this measurement [2, 3, 10, 27]. Previous work has also shown that if a test suite detects a large number of simple faults, caused by a single incorrect line of source code, it will detect a large number of harder, multi-line faults [28, 32]. This implies that if a test suite can kill a large proportion of mutants, it can also detect a large proportion of the more difficult faults in the software. The literature thus suggests that the mutant detection rate of a suite is a fairly good measurement of its fault detection ability. We will return to this issue in Sections 6 and 7.

We can now describe the remaining rows of Table 2. The seventh row shows how many mutants PIT generated for each project. The eighth row shows how many of those mutants could be detected by the suite. The ninth row shows how many of those mutants could not be detected by the entire test suite and were therefore assumed to be equivalent (i.e., row 7 is the sum of rows 8 and 9). The last row gives the equivalent mutants as a percentage of the total.

[4]Manual inspection is required because automatically determining whether a mutant is equivalent is undecidable [33].

3.4 Generating Test Suites

For each subject program, we used Java's reflection API to identify all of the test methods in the program's master suite. We then generated new test suites of fixed size by randomly selecting a subset of these methods without replacement. More concretely, we created a JUnit suite by repeatedly using the `TestSuite.addTest(Test t)` method. Each suite was created as a JUnit suite so that the necessary set-up and tear-down code was run for each test method. Given this procedure for creating suites, in this paper the size of our random suites should always be understood as the number of test methods they contain, i.e., the number of times `addTest` was called.

We made 1,000 suites of each of the following sizes: 3 methods, 10 methods, 30 methods, 100 methods, and so on, up to the largest number following this pattern that was less than the total number of test methods. This resulted in a total of 31,000 test suites across the five subject systems. Comparing a large number of suites from the same project allows us to control for size; it also allows us to apply our results to the common research practice of comparing test suites generated for the same subject program using different test generation methodologies.

3.5 Measuring Coverage

We used the open source tool CodeCover [8] to measure three types of coverage: statement, decision, and modified condition coverage. Statement coverage refers to the fraction of the executable statements in the program that are run by the test suite. It is relatively easy to satisfy, easy to understand and can be measured quickly, making it popular with developers. However, it is one of the weaker forms of coverage, since executing a line does not necessarily reveal an error in that line.

Decision coverage refers to the fraction of decisions (i.e., branches) in the program that are executed by its test suite. Decision coverage is somewhat harder to satisfy and measure than statement coverage.

Modified condition coverage (MCC) is the most difficult of these three to satisfy. For a test suite to be modified condition adequate, i.e., to have 100% modified condition coverage, the suite must include 2^n test cases for every decision with n conditions[5] in it [22]. This form of coverage is not commonly used in practice; however, it is very similar to mod-

[5]A condition is a boolean expression that cannot be decomposed into a simpler boolean expression. Decisions are composed of conditions and one or more boolean operators.

ified condition/decision coverage (MC/DC), which is widely used in the avionics industry. Specifically, Federal Aviation Administration standard DO-178B states that the most critical software in the aircraft must be tested with a suite that is modified condition/decision coverage adequate [22]. MC/DC is therefore one of the most stringent forms of coverage that is widely and regularly used in practice. Measuring modified condition coverage provides insight into whether stronger coverage types such as MCC and MC/DC provide practical benefits that outweigh the extra cost associated with writing enough tests to satisfy them.

We did not measure any type of dataflow coverage, since very few tools for Java can measure these types of coverage. One exception is Coverlipse [9], which can measure all-use coverage but can only be used as an Eclipse plugin. To the best of our knowledge, there are no open source coverage tools for Java that can measure other data flow coverage criteria or that can be used from the command line. Since developers use the tools they have, they are unlikely to use dataflow coverage metrics. Using the measurements that developers use, whether due to tool availability or legal requirements, means that our results will more accurately reflect current development practice. However, we plan to explore dataflow coverage in future work to determine if developers would benefit from using these coverage types instead.

3.6 Measuring Effectiveness

We used two effectiveness measurements in this study: the *raw effectiveness measurement* and the *normalized effectiveness measurement*. The raw kill score is the number of mutants a test suite detected divided by the total number of non-equivalent mutants that were generated for the subject program under test. The normalized effectiveness measurement is the number of mutants a test suite detected divided by the number of non-equivalent mutants it covers. A test suite covers a mutant if the mutant was made by altering a line of code that is executed by the test suite, implying that the test suite can potentially detect the mutant.

We included the normalized effectiveness measurement in order to compare test suites on a more even footing. Suppose we are comparing suite A, with 50% coverage, to suite B, with 60% coverage. Suite B will almost certainly have a higher raw effectiveness measurement, since it covers more code and will therefore almost certainly kill more mutants. However, if suite A kills 80% of the mutants that it covers, while suite B kills only 70% of the mutants that it covers, suite A is in some sense a better suite. The normalized effectiveness measurement captures this difference. Note that it is possible for the normalized effectiveness measurement to drop when a new test case is added to the suite if the test case covers a lot of code but kills few mutants.

It may be helpful to think of the normalized effectiveness measurement as a measure of depth: how thoroughly does the test suite exercise the code that it runs? The raw effectiveness measurement is a measure of breadth: how much code does the suite exercise?

Note that the number of non-equivalent mutants covered by a suite is the maximum number of mutants the suite could possibly detect, so the normalized effectiveness measurement ranges from 0 to 1. The raw effectiveness measurement, in general, does not reach 1, since most suites kill a small percentage of the non-equivalent mutants. However, note that the full test suite has both a normalized effectiveness measurement of 1 and a raw effectiveness measurement of 1, since we decided that any mutants it did not kill are equivalent.

4. RESULTS

In this section, we quantitatively answer the three research questions posed in Section 1. As Section 3 explained, we collected the data to answer these questions by generating test suites of fixed size via random sampling; measuring their statement, decision and MCC coverage with CodeCover; and measuring their effectiveness with the mutation testing tool PIT.

4.1 Is Size Correlated With Effectiveness?

Research Question 1 asked if the effectiveness of a test suite is influenced by the number of test methods it contains. This research question provides a "sanity check" that supports the use of the effectiveness metric. Figure 2 shows some of the data we collected to answer this question. In each subfigure, the x axis indicates suite size on a logarithmic scale while the y axis shows the range of normalized effectiveness values we computed. The red line on each plot was fit to the data with R's `lm` function[6]. The adjusted r^2 value for each regression line is shown in the bottom right corner of each plot. These values range from 0.26 to 0.97, implying that the correlation coefficient r ranges from 0.51 to 0.98. This indicates that there is a moderate to very high correlation between normalized effectiveness and size for these projects[7]. The results for the non-normalized effectiveness measurement are similar, with the r^2 values ranging from 0.69 to 0.99, implying a high to very high correlation between non-normalized effectiveness and size. The figure for this measurement can be found online[8].

> ANSWER 1. *Our results suggest that, for large Java programs, there is a moderate to very high correlation between the effectiveness of a test suite and the number of test methods it contains.*

4.2 Is Coverage Correlated With Effectiveness When Size Is Ignored?

Research Question 2 asked if the effectiveness of a test suite is correlated with the coverage of the suite when we ignore the influence of suite size. Tables 3 and 4 show the Kendall τ correlation coefficients we computed to answer this question; all coefficients are significant at the 99.9% level[9]. Table 3

[6]Size and the logarithm of size were used as the inputs.

[7]Here we use the Guildford scale [21] for verbal description, in which correlations with absolute value less than 0.4 are described as "low", 0.4 to 0.7 as "moderate", 0.7 to 0.9 as "high", and over 0.9 as "very high".

[8]`http://linozemtseva.com/research/2014/icse/coverage/`

[9]Kendall's τ is similar to the more common Pearson coefficient but does not assume that the variables are linearly related or that they are normally distributed. Rather, it measures how well an arbitrary monotonic function could fit the data. A high correlation therefore means that we can predict the rank order of the suites' effectiveness values given the rank order of their coverage values, which in practice is nearly as useful as predicting an absolute effectiveness score. We used it instead of the Pearson coefficient to avoid introducing unnecessary assumptions about the distribution of the data.

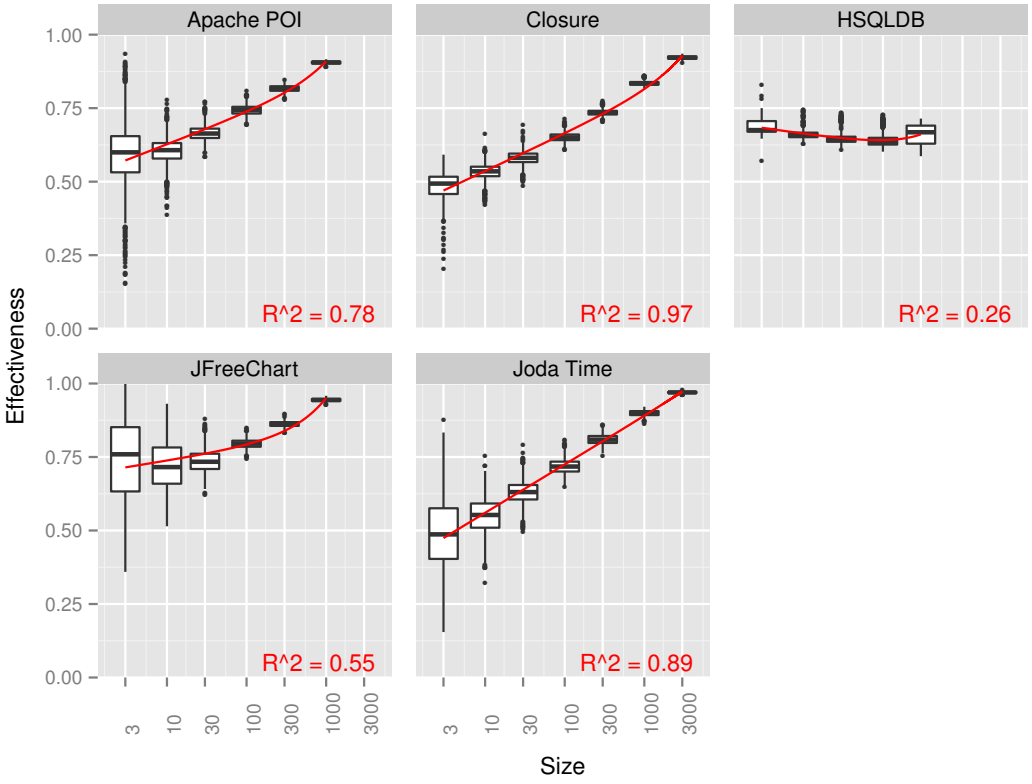

Figure 2: Normalized effectiveness scores plotted against size for all subjects. Each box represents the 1000 suites of a given size that were created from a given master suite.

gives the correlation between the different coverage types and the normalized effectiveness measurement. Table 4 gives the correlation between the different coverage types and the non-normalized effectiveness measurement. For all projects but HSQLDB, we see a moderate to very high correlation between coverage and effectiveness when size is not taken into account. HSQLDB is an interesting exception: when the effectiveness measurement is normalized by the number of covered mutants, there is a low *negative* correlation between coverage and effectiveness. This means that the suites with higher coverage kill fewer mutants per unit of coverage; in other words, the suites with higher coverage contain test cases that run a lot of code but do not kill many mutants in that code. Of course, since the suites kill more mutants in total as they grow, there is a positive correlation between coverage and non-normalized effectiveness for HSQLDB.

> ANSWER 2. *Our results suggest that, for many large Java programs, there is a moderate to high correlation between the effectiveness and the coverage of a test suite when the influence of suite size is ignored. Research Question 3 explores whether this correlation is caused by the larger size of the suites with higher coverage.*

4.3 Is Coverage Correlated With Effectiveness When Size Is Fixed?

Research Question 3 asked if the effectiveness of a test suite is correlated with its coverage when the number of test cases in the suite is controlled for. Figure 3 shows the data we collected to answer this question. Each panel shows

Table 3: The Kendall τ correlation between normalized effectiveness and different types of coverage when suite size is ignored. All entries are significant at the 99.9% level.

Project	Statement	Decision	Mod. Cond.
Apache POI	0.75	0.76	0.77
Closure	0.83	0.83	0.84
HSQLDB	−0.35	−0.35	−0.35
JFreeChart	0.50	0.53	0.53
Joda Time	0.80	0.80	0.80

Table 4: The Kendall τ correlation between non-normalized effectiveness and different types of coverage when suite size is ignored. All entries are significant at the 99.9% level.

Project	Statement	Decision	Mod. Cond.
Apache POI	0.94	0.94	0.94
Closure	0.95	0.95	0.95
HSQLDB	0.81	0.80	0.79
JFreeChart	0.91	0.95	0.92
Joda Time	0.85	0.85	0.85

the results we obtained for one project and one suite size. The project name is given at the top of each column, while the suite size is given at the right of each row. Different coverage types are differentiated by colour. The bottom row is a margin plot that shows the results for all sizes, while the rightmost column is a margin plot that shows the results for

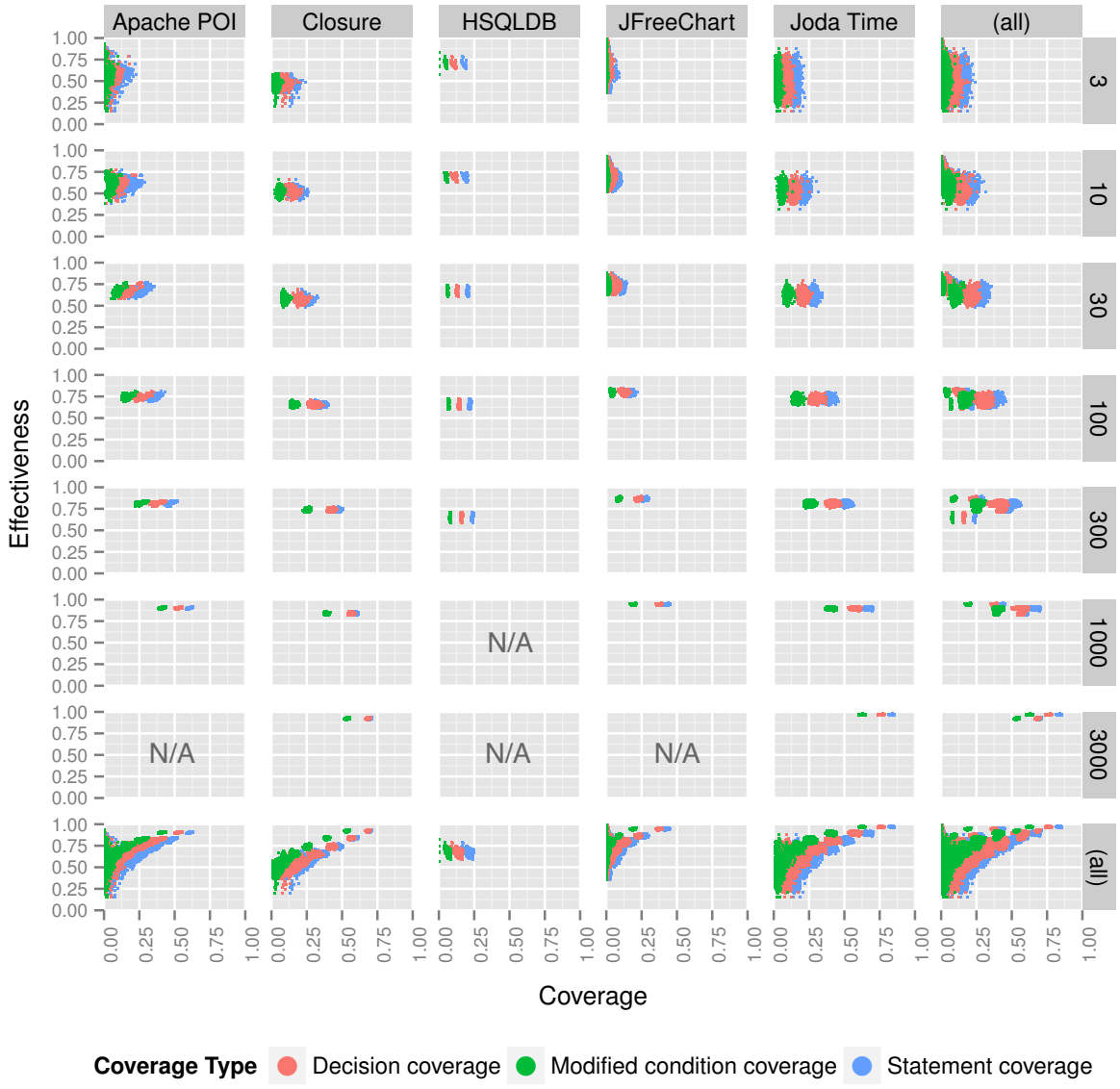

Figure 3: Normalized effectiveness scores (left axis) plotted against coverage (bottom axis) for all subjects. Rows show the results for one suite size; columns show the results for one project. N/A indicates that the project did not have enough test cases to fill in that frame.

all projects. The figure shows the results for the normalized effectiveness measurement; the non-normalized effectiveness measurements tend to be small and difficult to see at this size. The figure for the non-normalized effectiveness measurement can be found online with the other supplementary material.

We computed the Kendall τ correlation coefficient between effectiveness and coverage for each project, each suite size, each coverage type, and both effectiveness measures. Since this resulted in a great deal of data, we summarize the results here; the full dataset can be found on the same website as the figures.

Our results were mixed. Controlling for suite size always lowered the correlation between coverage and effectiveness. However, the magnitude of the change depended on the effectiveness measurement used. In general, the normalized effectiveness measurements had low correlations with coverage once size was controlled for while the non-normalized effectiveness measurements had moderate correlations with coverage once size was controlled for.

That said, the results varied by project. Joda Time was at one extreme: the correlation between coverage and effectiveness ranged from 0.80 to 0.85 when suite size was ignored, but dropped to essentially zero when suite size was controlled for. The same effect was seen for Closure when the normalized effectiveness measurement was used.

Apache POI fell at the other extreme. For this project, the correlation between coverage and the non-normalized effectiveness measurement was 0.94 when suite size was ignored, but dropped to a range of 0.46 to 0.85 when suite size was controlled for. While this is in some cases a large drop, a correlation in this range can provide useful information about the quality of a test suite.

A very interesting result is that, in general, the coverage type used did not have a strong impact on the results. This is true even though the effectiveness scores (y values) for each suite are the same for all three coverage types (x values). To clarify this, consider Figure 4. The figure shows two hypothetical graphs of effectiveness against coverage. In the top graph, coverage type 1 is not strongly correlated with effectiveness. In the bottom graph, coverage type 2 *is* strongly correlated with effectiveness even though the y-value of each point has not changed (e.g., the triangle is at $y = 0.8$ in both graphs). We do *not* see this difference between statement, decision, and MCC coverage, suggesting that the different types of coverage are measuring the same thing. We can confirm this intuition by measuring the correlation between different coverage types for each suite (Table 5). Given these high correlations, and given that the shape of the point clouds are similar for all three coverage measures (see Figure 3), we can conclude that the coverage type used has little effect on the relationship between coverage and effectiveness in this study.

Table 5: The Kendall τ and Pearson correlations between different types of coverage for all suites from all projects.

Coverage Types	Tau	Pearson
Statement/Decision	0.92	0.99
Decision/MCC	0.91	0.98
Statement/MCC	0.92	0.97

ANSWER 3. *Our results suggest that, for large Java programs, the correlation between coverage and effectiveness drops when suite size is controlled for. After this drop, the correlation typically ranges from low to moderate, meaning it is not generally safe to assume that effectiveness is correlated with coverage. The correlation is stronger when the non-normalized effectiveness measurement is used. Additionally, the type of coverage used had little influence on the strength of the relationship.*

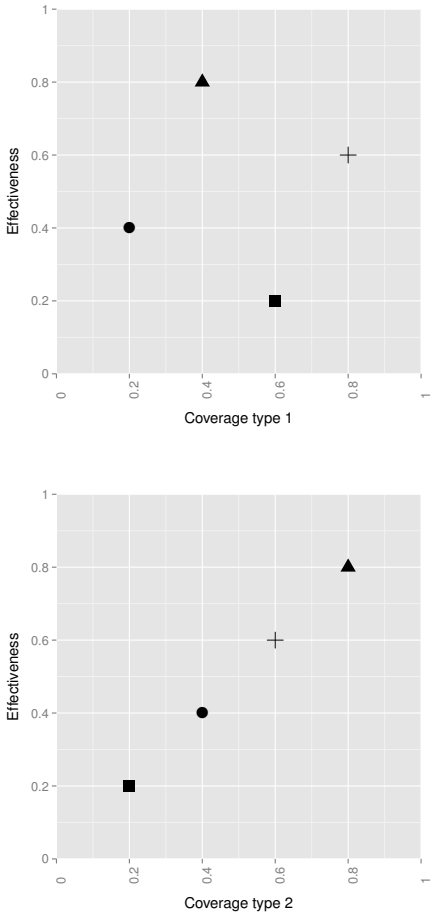

Figure 4: Hypothetical graphs of effectiveness against two coverage types for four test suites. The top graph shows a coverage type that is not correlated with effectiveness; the bottom graph shows a coverage type that is correlated with effectiveness.

5. DISCUSSION

The goal of this work was to determine if a test suite's coverage is correlated with its fault detection effectiveness when suite size is controlled for. We found that there is typically a moderate to high correlation between coverage and effectiveness when suite size is ignored, and that this drops to a low to moderate correlation when size is controlled. This result suggests that coverage alone is not a good predictor of test suite effectiveness; in many cases, the apparent relationship is largely due to the fact that high coverage suites contain more test cases. The results for Joda Time and Closure, in particular, demonstrate that it is not safe in general to assume that coverage is correlated with effectiveness. Interestingly, the suites for Joda Time and Closure are the largest and most comprehensive of the five suites we studied, which might indicate that the correlation becomes weaker as the suite improves.

In addition, we found that the type of coverage measured had little impact on the correlation between coverage and effectiveness. This is reinforced by the shape of the point clouds in Figure 3: for any one project and suite size, the clouds corresponding to the three coverage types are similar in shape and size. This, in combination with the high correlation between different coverage measurements, suggests that stronger coverage types provide little extra information about the quality of the suite.

Our findings have implications for developers, researchers, and standards bodies. Developers may wish to use this information to guide their use of coverage. While coverage measures are useful for identifying under-tested parts of a program, and low coverage may indicate that a test suite is inadequate, high coverage does not indicate that a test suite is effective. This means that using a fixed coverage value as a quality target is unlikely to produce an effective test suite. While members of the testing community have previously made this point [13, 30], it has been difficult to evaluate their suggestions due to a lack of studies that considered systems of the scale that we investigated. Additionally, it may be in the developer's best interest to use simpler coverage measures. These measures provide a similar amount of information about the suite's effectiveness but introduce less measurement overhead.

Researchers may wish to use this information to guide their tool-building. In particular, test generation techniques often attempt to maximize the coverage of the resulting suite; our results suggest that this may not be the best approach.

Finally, our results are pertinent to standards bodies that set requirements for software testing. The FAA standard DO-178B, mentioned earlier in this paper, requires the use of MC/DC adequate suites to ensure the quality of the resulting software; however, our results suggest that this requirement may increase expenses without necessarily increasing quality.

Of course, developers still want to measure the quality of their test suites, meaning they need a metric that *does* correlate with fault detection ability. While this is still an open problem, we currently feel that mutation score may be a good substitute for coverage in this context [27].

6. THREATS TO VALIDITY

In this section, we discuss the threats to the construct validity, internal validity, and external validity of our study.

6.1 Construct Validity

In our study we measured the size, coverage and effectiveness of random test suites. Size and coverage are straightforward to measure, but effectiveness is more nebulous, as we are attempting to predict the fault-detection ability of a suite that has never been used in practice. As we described in Section 3.3, previous and current work suggests that a suite's ability to kill mutants is a fairly good measurement of its ability to detect real faults [2, 3, 10, 27]. This suggests that, in the absence of equivalent mutants, this metric has high construct validity. Unfortunately, our treatment of equivalent mutants introduces a threat to the validity of this measurement. Recall that we assumed that any mutant that could not be detected by the program's entire test suite is equivalent. This means that we classified up to 35% of the generated mutants as equivalent (see the final row of Table 2). In theory, these mutants are a random subset of the entire set of mutants, so ignoring them should not affect our results. However, this may not be true. For example, if the developers frequently test for off-by-one errors, mutants that simulate this error will be detected more often and will be less likely to be classified as equivalent.

6.2 Internal Validity

Our conclusions about the relationship between size, coverage and effectiveness depend on our calculations of the Kendall τ correlation coefficient. This introduces a threat to the internal validity of the study. Kendall's original formula for τ assumes that there are no tied ranks in the data; that is, if the data were sorted, no two rows could be exchanged without destroying the sorted order. When ties do exist, two issues arise. First, since the original formula does not handle ties, a modified one must be used. We used the version proposed by Adler [1]. Second, ties make it difficult to compute the statistical significance of the correlation coefficient. It it possible to show that, in the absence of ties, τ is normally distributed, meaning we can use Z-scores to evaluate significance in the usual way. However, when ties are present, the distribution of τ changes in a way that depends on the number and nature of the ties. This can result in a non-normal distribution [18]. To determine the impact of ties on our calculations, we counted both the number of ties that occurred and the total number of comparisons done to compute each τ. We found that ties rarely occurred: for the worst calculation, 4.6% of the comparisons resulted in a tie, but for most calculations this percentage was smaller by several orders of magnitude. Since there were so few ties, we have assumed that they had a negligible effect on the normal distribution.

Another threat to internal validity stems from the possibility of duplicate test suites: our results might be skewed if two or more suites contain the same subset of test methods. Fortunately, we can evaluate this threat using the information we collected about ties: since duplicate suites would naturally have identical coverage and effectiveness scores, the number of tied comparisons provides an upper bound on how many identical suites were compared. Since the number of ties was so low, the number of duplicate suites must be similarly low, and so we have ignored the small skew they may have introduced to avoid increasing the memory requirements of our study unnecessarily.

Since we have studied correlations, we cannot make any claims about the direction of causality.

6.3 External Validity

There are six main threats to the external validity of our study. First, previous work suggests that the relationship between size, coverage and effectiveness depends on the difficulty of detecting faults in the program [3]. Furthermore, some of the previous work was done with hand-seeded faults, which have been shown to be harder to detect than both mutants and real faults [2]. While this does not affect our results, it does make it harder to compare them with those of earlier studies.

Second, some of the previous studies found that a relationship between coverage and effectiveness did not appear until very high coverage levels were reached [14,17,24]. Since the coverage of our generated suites rarely reached very high values, it is possible that we missed the existence of such a relationship. That said, it is not clear that such a relationship would be useful in practice. It is very difficult to reach extremely high levels of coverage, so a relationship that does not appear until 90% coverage is reached is functionally equivalent to no relationship at all for most developers.

Third, in object-oriented systems, most faults are usually found in just a few of the system's components [12]. This means that the relationship between size, coverage and effectiveness may vary by class within the system. It is therefore possible that coverage is correlated with effectiveness in classes with specific characteristics, such as high churn. However, our conclusions still hold for the common practice of measuring the coverage of a program's entire test suite.

Fourth, there may be other features of a program or a suite that affect the relationship between coverage and effectiveness. For example, previous work suggests that the size of a class can affect the validity of object-oriented metrics [11]. While we controlled for the size of each test suite in this study, we did not control for the size of the class that each test method came from.

Fifth, as discussed in Section 3.2, our subjects had to meet certain inclusion criteria. This means that they are fairly similar, so our results may not generalize to programs that do not meet these criteria. We attempted to mitigate this threat by selecting programs from different application domains, thereby ensuring a certain amount of variety in the subjects. Unfortunately, it was difficult to find acceptable

subjects; in particular, the requirement that the subjects have 1,000 test cases proved to be very difficult to satisfy. In practice, it seems that most open source projects do not have comprehensive test suites. This is supported by Gopinath et al.'s study [20], where only 729 of the 1,254 open source Java projects they initially considered, or 58%, had test suites at all, much less comprehensive suites.

Finally, while our subjects were considerably larger than the programs used in previous studies, they are still not large by industrial standards. Additionally, all of the projects were open source, so our results may not generalize to closed source systems.

7. FUTURE WORK

Our next step is to confirm our findings using real faults to eliminate this threat to validity. We will also explore dataflow coverage to determine if these coverage types are correlated with effectiveness.

It may also be helpful to perform a longitudinal study that considers how the coverage and effectiveness of a program's test suite change over time. By cross-referencing coverage information with bug reports, it might be possible to isolate those bugs that were covered by the test suite but were not immediately detected by it. Examining these bugs may provide insight into which bugs are the most difficult to detect and how we can improve our chances of detecting them.

8. CONCLUSION

In this paper, we studied the relationship between the number of methods in a program's test suite, the suite's statement, decision, and modified condition coverage, and the suite's mutant effectiveness measurement, both normalized and non-normalized. From the five large Java programs we studied, we drew the following conclusions:

- In general, there is a low to moderate correlation between the coverage of a test suite and its effectiveness when its size is controlled for.
- The strength of the relationship varies between software systems; it is therefore not generally safe to assume that effectiveness is strongly correlated with coverage.
- The type of coverage used had little impact on the strength of the correlation.

These results imply that high levels of coverage do not indicate that a test suite is effective. Consequently, using a fixed coverage value as a quality target is unlikely to produce an effective test suite. In addition, complex coverage measurements may not provide enough additional information about the suite to justify the higher cost of measuring and satisfying them.

9. REFERENCES

[1] L. M. Adler. A modification of Kendall's tau for the case of arbitrary ties in both rankings. *Journal of the American Statistical Association*, 52(277), 1957.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the Int'l Conf. on Soft. Eng.*, 2005.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Soft. Eng.*, 32(8), 2006.

[4] Apache POI. `http://poi.apache.org`.

[5] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proc. of the Int'l Symposium on Software Reliability Engineering*, 1999.

[6] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proc. of the Int'l Workshop on Advances in Model-Based Testing*, 2005.

[7] Closure Compiler. `https://code.google.com/p/closure-compiler/`.

[8] CodeCover. `http://codecover.org/`.

[9] Coverlipse. `http://coverlipse.sourceforge.net/`.

[10] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proc. of the Int'l Symposium on Software Testing and Analysis*, 1996.

[11] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Soft. Eng.*, 27(7), 2001.

[12] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Soft. Eng.*, 26(8), 2000.

[13] M. Fowler. Test coverage. `http://martinfowler.com/bliki/TestCoverage.html`, 2012.

[14] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proc. of the Int'l Symposium on Foundations of Soft. Eng.*, 1998.

[15] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proc. of the Symposium on Testing, Analysis, and Verification*, 1991.

[16] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Soft. Eng.*, 19(8), 1993.

[17] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3), 1997.

[18] J. D. Gibbons. *Nonparametric Measures of Association*. Sage Publications, 1993.

[19] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proc. of the Int'l Symp. on Soft. Testing and Analysis*, 2013.

[20] R. Gopinath, C. Jenson, and A. Groce. Code coverage for suite evaluation by developers. In *Proc. of the Int'l Conf. on Soft. Eng.*, 2014.

[21] J. P. Guilford. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, 1942.

[22] K. Hayhurst, D. Veerhusen, J. Chilenski, and L. Rierson. A practical tutorial on modified condition/decision coverage. Technical report, NASA Langley Research Center, 2001.

[23] HSQLDB. `http://hsqldb.org`.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the*

Int'l Conf. on Soft. Eng., 1994.

[25] JFreeChart. `http://jfree.org/jfreechart`.

[26] Joda Time. `http://joda-time.sourceforge.net`.

[27] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? Technical Report UW-CSE-14-02-02, University of Washington, March 2014.

[28] K. Kapoor. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Soft. Eng.*, 2(2), 2006.

[29] E. Kit. *Software Testing in the Real World: Improving the Process*. ACM Press, 1995.

[30] B. Marick. How to misuse code coverage. `http://www.exampler.com/testing-com/writings/coverage.pdf`, 1997.

[31] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. of the Int'l Symposium on Software Testing and Analysis*, 2009.

[32] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Soft. Eng. and Methodology*, 1(1), 1992.

[33] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *Proc. of the Conf. on Computer Assurance*, 1996.

[34] W. Perry. *Effective Methods for Software Testing*. Wiley Publishing, 2006.

[35] PIT. `http://pitest.org/`.

[36] Randoop. `https://code.google.com/p/randoop/`.

[37] R. Sharma. Guidelines for coverage-based comparisons of non-adequate test suites. Master's thesis, University of Illinois at Urbana-Champaign, 2013.

[38] SLOCCount. `http://dwheeler.com/sloccount`.

[39] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proc. of the Int'l Symposium on Software Reliability Engineering*, 1994.

Empir Software Eng (2014) 19:378–417
DOI 10.1007/s10664-013-9267-7

Effectiveness for detecting faults within and outside the scope of testing techniques: an independent replication

Cecilia Apa · Oscar Dieste · Edison G. Espinosa G. · Efraín R. Fonseca C.

Published online: 8 August 2013

Abstract The verification and validation activity plays a fundamental role in improving software quality. Determining which the most effective techniques for carrying out this activity are has been an aspiration of experimental software engineering researchers for years. This paper reports a controlled experiment evaluating the effectiveness of two unit testing techniques (the functional testing technique known as *equivalence partitioning* (EP) and the control-flow structural testing technique known as *branch testing* (BT)). This experiment is a literal replication of Juristo et al. (2013). Both experiments serve the purpose of determining whether the effectiveness of BT and EP varies depending on whether or not the faults are visible for the technique (InScope or OutScope, respectively). We have used the materials, design and procedures of the original experiment, but in order to adapt the experiment to the context we have: (1) reduced the number of studied techniques from 3 to 2; (2) assigned subjects to experimental groups by means of stratified randomization to balance the influence of programming experience; (3) localized the experimental

Communicated by: Jeffrey C. Carver, Natalia Juristo, Teresa Baldassarre and Sira Vegas.

C. Apa
Universidad de la República, Julio Herrera y Reissig 565, Montevideo, Uruguay
e-mail: ceapa@fing.edu.uy

O. Dieste
Universidad Politécnica de Madrid, Boadilla del Monte 28660, Madrid, Spain
e-mail: odieste@fi.upm.es

E. G. Espinosa G.
Escuela Politécnica del Ejército Sede Latacunga, Latacunga, Ecuador
e-mail: egespinosa1@espe.edu.ec

E. R. Fonseca C. (✉)
Escuela Politécnica del Ejército, Sangolquí, Ecuador
e-mail: erfonseca@espe.edu.ec

materials and (4) adapted the training duration. We ran the replication at the Escuela Politécnica del Ejército Sede Latacunga (ESPEL) as part of a software verification & validation course. The experimental subjects were 23 master's degree students. EP is more effective than BT at detecting InScope faults. The session/program and group variables are found to have significant effects. BT is more effective than EP at detecting OutScope faults. The session/program and group variables have no effect in this case. The results of the replication and the original experiment are similar with respect to testing techniques. There are some inconsistencies with respect to the group factor. They can be explained by small sample effects. The results for the session/program factor are inconsistent for InScope faults. We believe that these differences are due to a combination of the fatigue effect and a technique x program interaction. Although we were able to reproduce the main effects, the changes to the design of the original experiment make it impossible to identify the causes of the discrepancies for sure. We believe that further replications closely resembling the original experiment should be conducted to improve our understanding of the phenomena under study.

Keywords Replication · Experiment · Unit testing · Reporting guidelines

1 Introduction

Verification and validation (V&V) activities play a fundamental role in improving software quality. There are many approaches for carrying out V&V, but we do not know for certain which technique or combination of techniques is more effective for each type of software validation: unit, integration or system testing.

Determining the effectiveness of unit testing techniques soon attracted the attention of experimental software engineering (SE) researchers. Way back in 1978, Myers compared the effectiveness of functional and structural testing techniques (Myers 1978). Soon after, Basili and Selby studied the effectiveness of functional and structural techniques and code reading (Basili and Selby 1985), and started up a much replicated family of experiments.

In this paper, we report the replication of a controlled experiment belonging to Basili's family. The original experiment was conducted at the Universidad Politécnica de Madrid (UPM) in December 2005. According to Gómez et al. (2010) and Gómez (2012), this replication can be classified as a literal (that is, the replication resembles the original experiment as closely as possible), joint (some of the original experimenters participated in the replication) and external (the replication was conducted at a different site) replication of the original experiment. The replication was conducted at the Escuela Politécnica del Ejército Sede Latacunga (ESPEL) in Ecuador in December 2011.

The purpose of conducting the replication was to verify the results of the original experiment, and, in the event of inconsistencies, identify which factors or parameters could explain the differences between the experiments. The alternative, if this were not possible, would be to conduct further replications. Another goal of this paper was to use Carver's guidelines (Carver 2010) and evaluate their strengths for reporting replications.

Following Carver's guidelines, the paper describes information about the original study in Section 2, details information about the replication in Section 3, compares

the replication results to the original results in Section 4 and, finally, reports the conclusions across studies in Section 5.

2 Information About the Original Study

The original experiment was carried out by Juristo et al. (2013). Its aim was to study how effective structural and functional testing techniques are at detecting faults. Juristo et al.'s (2013) experiment is, in turn, a differentiated replication of the family of experiments started by Basili and Selby (1985, 1987) in 1982, and later replicated by several researchers like Kamsties and Lott (1995) and Wood et al. (1997). The main difference between the original experiment and Basili's family of experiments lies in the way in which the faults were seeded in the programs.

Basili, Selby and others referred to the types of faults used in their experiments as fairly representative of the defects that tend to occur in software development (Basili and Selby 1987). Therefore, those experiments evaluated how effective the testing process is expected to be in practice. However, the fault types used in Juristo et al.'s (2013) experiment were simpler. They can be divided into two major categories: faults that can and faults that, at least in theory, cannot be detected by the structural or functional test case generation strategy (InScope and OutScope faults, respectively).

The differentiation of the two fault types is useful for studying the effectiveness of the testing techniques more precisely than before, as it helps to distinguish the effect of the actual technique (that is, the detection of InScope faults) from the positive effects that the technique has on the tester but that cannot be attributed to the technique per se (that is, OutScope faults). Both effects were confounded in previous experiments.

2.1 Research Questions

The original experiment report does not include an explicit research question. However, this question can be easily inferred from the experimental design. It can be stated in GQM (Basili 1992) as follows:

Analyse the application of software testing techniques for the purpose of finding out how effective they are at unit testing level with respect to different fault types (InScope, OutScope) from the point of view of testers in the context of a controlled experiment in academia.

This research question is useful for identifying the key elements of the original experiment.

(A) **Main Factor:** The original experiment studied two testing techniques: the functional testing technique known as *equivalence partitioning* (EP) and the control-flow structural testing technique known as *branch testing* (BT).
The original experiment also tests another technique: code reading by stepwise abstraction technique (CR). As CR is capable of detecting all fault types, it is used in the original experiment for the purpose of control not as a main factor. According to the original experimenters (Juristo et al. 2013), *"As a control group, we have used a technique* (the CR technique) *with a strategy capable, at least in theory, of detecting all program faults"*.

(B) **Response Variables:** Technique effectiveness was measured as the percentage of faults located by the techniques over the total seeded faults. As the research question is concerned with **InScope** and **OutScope** faults, the effectiveness of the techniques was calculated separately for each fault type.

(C) **Hypotheses:**

H_{10}: There is no difference in the effectiveness of EP, BT and CR with respect to the detection of faults within their scope.

H_{11}: The effectiveness of EP, BT and CR differs with respect to faults within their scope.

H_{20}: There is no difference in the effectiveness of EP, BT and CR with respect to the detection of faults outside their scope.

H_{21}: The effectiveness of EP, BT and CR differs with respect to faults outside their scope.

2.2 Participants

The subjects participating in the experiment were 46 Universidad Politécnica de Madrid undergraduate computing engineering students. The students were taking the 4th-year software verification and validation course as part of their five-year degree programme. Students had little or no professional experience of software development.

2.3 Design

As a between-subjects design with a total of 46 experimental subjects would result in very few subjects per group ($46/3 \simeq 15$) and low statistical power, the authors of the original study used a within-subjects design, where each subject applied each of the tested techniques at three different times (sessions). Sessions had no set time limit, and each session lasted on average four hours. Table 1 summarizes the experimental design.

Although the within-subjects design increases statistical power, it also poses several validity threats. For example, carryover, learning effects (maturation) and fatigue are a possibility:

(a) **Carryover:** The residual effect that administering one treatment to a subject has on another treatment administered later to the same subject, where the residual

Table 1 Original experiment design

Program	Cmdline			Ntree			Nametbl		
Session	Session 1			Session 2			Session 3		
Techniq.	CR	BT	EP	CR	BT	EP	CR	BT	EP
Group 1	X	–	–	–	X	–	–	–	X
Group 2	X	–	–	–	–	X	–	X	–
Group 3	–	X	–	–	–	X	X	–	–
Group 4	–	X	–	X	–	–	–	–	X
Group 5	–	–	X	X	–	–	–	X	–
Group 6	–	–	X	–	X	–	X	–	–

effect increases or decreases the effectiveness of the later treatment, is known as carryover (Brown 1980). Carryover is an important risk in medical experiments, as drug residues can remain in the body for quite some time and interact with later treatments (Senn 2002). It is harder to imagine what underlying cause could produce a carryover effect in SE. However, carryover has been explicitly cited in the SE literature (Kitchenham et al. 2003) and hence should be taken into account.

(b) **Learning:** Subject performance may increase irrespective of the applied treatments as a result of practice acquired in successive experimental sessions.

(c) **Fatigue:** Subject performance may drop as a result of fatigue caused by applying treatments at short intervals in successive experimental sessions.

To minimize all these threats, the sessions were held one week apart. Even so, as shown in Table 1, the original experimenters added the session and group variables to the design in order to determine whether such threats materialize. Specifically, according to the original experimenters, the session variable can identify the presence of learning or fatigue effects, and the group variable can detect carryover effects.

2.4 Artefacts

Several artefacts were used to operationalize the original experimental design:

- Training materials to improve or consolidate subject knowledge of the techniques under study
- Experimental objects on which to apply techniques
- Experimental materials to support experimental task performance.

The above-mentioned artefacts are described in the following. They are all available at Juristo et al. (2013), with the exception of the programs and fault descriptions. This is meant to assure that students participating in the experiments are not acquainted with them beforehand. However, programs and fault descriptions are available from the original experimenters via email.

2.4.1 Training Materials

The material used to train subjects on the application of the software testing techniques under study includes:

- **Reference guide:** Document containing the theoretical foundations and practical exercises for training experimental subjects in the application of software testing techniques.
- **Slides:** Support material for the trainer containing a summary of the reference guide.
- **Training programs:** Material used to supplement the theoretical groundwork, focusing on acquainting experimental subjects with the experiment execution environment.

2.4.2 Experimental Objects

For testing technique application, there should be at least as many programs as sessions, seeded with the right faults, because subjects cannot test the same

program with the same faults twice. The programs and faults used in the original experiment were:

(A) **Programs:** Three programs written in C were used: **cmdline**, **nametbl** and **ntree**. These are the same programs that were used in experiments run by Kamsties and Lott (1995) or Roper et al. (1997). As these programs are likely to affect (increase or decrease) the effectiveness of techniques, the programs were considered as blocking variables for analysis purposes. These programs perform the following functions:

  (a) **Cmdline** parses a command line to determine whether it is valid. If it is, it displays a description of the input command line; if it is not, it specifies why it is incorrect. Note that the program does not execute any of the commands, but merely validates the input.
  (b) **Nametbl** reads and processes file commands to test a series of functions used to manage an abstract data type, specifically a particular programming language symbol table.
  (c) **Ntree** reads and processes file commands to test a series of functions used to manage an abstract data type, specifically an n-ary tree.

(B) **Faults:** As mentioned earlier, different fault types were seeded in the original experiment than were used in earlier experiments (e.g.: Kamsties and Lott 1995; Roper et al. 1997).
The foremost difference between the original experiment and earlier experiments is the InScope and OutScope fault categorization for EP or BT. An InScope fault is a fault that can be detected by a correctly applied technique; faults are classed as OutScope otherwise.
Thus, for example, *unimplemented parts of the specification,* is a possible example of a fault that EP can detect (InScope). EP is capable of revealing this fault type because it generates test cases for all program specifications, including non-implemented specifications. Alternatively, *code for functionalities that are not in the specification* (e.g. when a programmer has mistakenly written or copied code that implements functions not accounted for in the specification) is an example of a fault that BT can detect. Branch testing's strategy prescribes that test cases should be generated to cover all the alternatives of 100 % of the code decisions, in which case it should detect superfluous code.
In order to systematically seed programs with faults, faults would have to have been classified by technique sensitivity. However, the original experimenters were unable to find any such classification, and they therefore generated the necessary fault types, shown in Table 2.

Table 2 Fault types (FT)

FT	Description
1	Unimplemented specification
2	Specific test data for achieving coverage
3	Combination of invalid equivalence classes
4	Chosen combination of valid equivalence classes
5	Test data for combining classes
6	Implementation detail
7	Implementation of unspecified functionality

Each of the fault types proposed by Juristo et al. (2013) are further detailed below.

(a) **Unimplemented specification:** The program does not implement the code for a particular specification. The BT technique is unable to identify this fault type, whereas EP will generate test cases capable of revealing this type of faults.
(b) **Specific test data for achieving coverage:** The program does not cover all the values specified in the requirements. The BT technique protocol does not indicate which data to select to assure that the test cases cover code decisions, whereas EP will generate test cases capable of detecting this fault type.
(c) **Combination of invalid equivalence classes:** Faults entered in programs by adding code that does not comply with the program specification. BT is capable of detecting this fault type, but EP is not.
(d) **Chosen combination of valid equivalence classes:** Unnecessary functionalities added to the code, which are already covered by another program specification. BT is capable of detecting this fault type, whereas EP is not.
(e) **Test data for combining classes:** Coding faults caused by the inclusion of functionalities that are not stated in the specifications. EP is not always able to detect this fault type, because, although the EP strategy prescribes that test cases should cover all the identified equivalence classes, it does not state exactly which class data should be selected for the test case. On the other hand, BT will generate test cases to detect this fault type.
(f) **Implementation detail:** Faults entered by programmers as a result of the choice of programming strategy to comply with the program specification. EP is unable to find this fault type, whereas BT can generate test cases capable of discovering such faults.
(g) **Implementation of unspecified functionality:** The program implements a functionality that is not in the specification. The EP technique is unable to identify this fault type. On the other hand, if applied correctly, BT will reveal such faults.

The programs were each seeded with six faults of the seven fault types. Three of these faults (called F1-F3) were InScope faults for EP, and the other three (F4-F6) were InScope faults for BT. Remember that the faults that are InScope for one technique are OutScope for the other, and vice versa. Table 3 details the faults seeded in each program.

2.4.3 Experimental Materials

The material used to execute the experiment includes the program specifications, experimental data collection forms, source code listing with seeded faults, executable code with seeded faults and guidelines for executing the experiment.

2.5 Context Variables

The original experiment explicitly considered the following contextual variables:

- **Environment:** Academia
- **Subject type:** Undergraduate students
- **Experience:** Students with little or no professional software development experience

Table 3 Faults seeded in programs

Technique	FT	F1	F2	F3	F4	F5	F6
Cmdline							
EP	1	X		X			
	2		X				
BT	3				X		
	4						X
	5					X	
Nametbl							
EP	1	X					
	2		X	X			
BT	3					X	
	5						X
	7				X		
Ntree							
EP	2	X	X	X			
BT	3					X	
	5				X		
	6						X

- **Program type:** Small-sized programs (150–220 LOC), with cyclomatic complexities ranging from 21 to 61
- **Program language:** C.

2.6 Execution Procedure

The original experiment was executed in three clearly defined stages, which the original experimenters called pre-session, during-session and post-session.

2.6.1 Pre-session

The pre-session is the stage of the experiment during which experimental subjects receive training on how to apply the testing techniques. The materials to be used to execute the experiment are also prepared at this stage. These materials, available in Juristo et al. (2013), are as follows:

- Experimental objects
- Forms
- Guides

2.6.2 During-Session

This is the stage during which the experiment proper is executed. The subjects take their places, far enough apart so that they cannot copy, in a room equipped for the purpose. Subjects will have been randomly assigned to groups before the first session. This assures that experimenters deliver the right materials to subjects during the session.

The first phase of the experimental session is test case generation. To do this, the subjects applying EP are supplied with:

- Experimental object specification (specification of the cmdline, nametbl or ntree program)

– Data collection forms for recording the test case and the expected output of each test case.

For the BT technique, subjects are supplied with:

– Source code listing with seeded faults in order to generate the test cases
– Data collection forms for recording the test cases and the expected output of each test case.

Note that none of the experimental subjects are given executable code during the test case generation phase, as the subjects might be tempted to read the programs instead of generating test cases to detect faults. This would frustrate the purpose of the experiment, which is to test how effective testing techniques, not software testers, are at detecting faults.

After they have generated the test cases, they are given:

– Executable code with seeded faults
– Data collection forms for recording the observed output of each test case execution
– The program specifications in order to test the generated cases
– Data collection forms for recording the detected faults.

The experimental subjects are not allowed to add any other test cases once they have been given the executable code at the end of the test case generation stage. This is controlled by the person responsible for monitoring the experiment execution.

Finally, the subjects fill the respective form with the outputs observed during the execution of each test case. The program faults are inferred from the comparison of the observed outputs with the expected outputs.

2.6.3 Post-Session

During this stage, all the material used by the experimental subjects is collected, and the data collection forms containing the test cases designed by the experimental subjects are detached for analysis. To do this, the test cases generated by the subjects are compared with previously designed test case templates. This analysis reveals the faults discovered by each subject applying each technique. This is not a strict comparison protocol, and test cases containing some sort of formal error that, with a minor correction, would generally identify faults are rated positively.

2.7 Summary of Results

Repeated measures ANOVA (rANOVA) was used to analyse the results, where the technique (BT, EP, CR), the program (ntree, cmdline, nametbl) and the group (six groups, termed G1-G6) were considered as factors. The sessions were not explicitly considered as factors, as they were confounded with the programs. The session/program and technique factors were considered as within-subjects factors, whereas the group factor was treated as a between-subjects factor. The fitted model was strictly additive and took the form:

$$Y = TECHNIQUE + PROGRAM + GROUP + e$$

The results obtained by the original experimenters can be summarized as follows.

2.7.1 InScope Response Variable

The rANOVA provided significant results for all factors: technique, program/session and group:

(A) **Technique:** The significant rANOVA result leads to the null hypothesis being rejected. Pairwise comparison showed that the code reading technique was less effective than the *equivalence partitioning* and *branch testing* techniques. In actual fact, code reading detected approximately 54 % of faults, whereas branch testing and equivalence partitioning detected 67.67 and 78.70 % of the seeded faults, respectively.

(B) **Program/session:** The significant rANOVA result suggests that either the program or the session influences effectiveness, but, as they are confounded, there is no way of reliably telling which one really has a bearing. Although the program/session interaction could theoretically have an effect, it is unlikely to do so because there is no reason why a program should increase the effectiveness of a session or vice versa.
The pairwise comparison reveals that cmdline<nametbl<ntree (we use the "<"symbol to indicate that subjects are less effective for cmdline than for nametbl and so on for all other variables). Alternatively, S1<S3<S2. The differences are only significant for cmdline/S1 and ntree/S2. As the design type used confounds the session and program variables, there are three possible grounds for the observed results: learning, fatigue or the possibility of one program being easier to test than another. If there were a learning effect, we should observe S1<S2<S3. If there were a fatigue effect (very unlikely, however, as the sessions were held one week apart), we should find that S3<S2<S1. Consequently, the program rather than the session is more likely to have a bearing on effectiveness.

(C) **Group:** As already mentioned, the group factor was introduced to detect the presence of carryover, as the original experiment had a within-subjects design. Pairwise comparison reveals significant differences of effectiveness among the sequences EP-CR-BT and EP-BT-CR and BT-CR-EP with respect to the order in which the techniques were applied. With a fault detection rate of 82.41 %, the EP-CR-BT sequence was more effective than the EP-BT-CR and BT-CR-EP sequences with rates of 55.55 and 57.41 %, respectively. As there are some significant differences between some levels of this factor but not between others, the original experimenters claimed that there does not appear to be any carryover effect from one technique to another.

2.7.2 OutScope Response Variable

The rANOVA provided significant results for all the factors: technique, program/session and group.

(A) **Technique:** The significant rANOVA result leads to the null hypothesis being rejected. Pairwise comparison shows that the *equivalence partitioning* technique was less effective (14.12 %) than the *branch testing* technique (29.09 %).

(B) **Program/session:** In this case again, the rANOVA result is significant and suggests that either the program or the session influences effectiveness. Because the program and session are confounded, there is no way of reliably telling which has a bearing on effectiveness.

The pairwise comparison shows that cmdline<nametbl (or, in session terms, S1<S3). Ntree (or S2) does not have significant differences with respect to the other two programs (sessions). This difference of effectiveness between S1 and S3 could be attributed to the session and not to the program. However, the results on technique effectiveness for Inscope faults, plus the fact that there are no statistical differences between S3 and S2, led the original experimenters to conclude that it is the program and not the session that makes the difference. As for the InScope faults, there would again be no learning effect.

(C) **Group:** Pairwise comparison reveals significant differences of effectiveness among sequences EP-CR-BT (8.33 %), and CR-BT-EP and EP-BT-CR (35.42 and 36.11 %, respectively) only with respect to the order in which the techniques were applied. As with the InScope variable, the original experimenters claimed that there does not appear to be any carryover effect improving technique effectiveness for faults outside their scope due to the order of technique application.

3 Information About the Replication

The replication was conducted at the Escuela Politécnica del Ejército sede Latacunga (ESPEL), Ecuador, with Master in Software Engineering students taking a software verification and validation course. The duration of this course is 80 h divided into seven, 10-h face-to-face sessions and a 10-h off-campus period set aside for homework and academic administrative matters. The face-to-face sessions were divided into three stages. In the first stage, the first part of the theoretical groundwork of the training was taught across three consecutive sessions. This was followed by a three-day break. Then the second stage (two consecutive sessions) covered the second theoretical part and training exercises. The third stage was the replication, which was run across two consecutive sessions as of the following day. The replication was one of the course assessment tests that carried most weight. This was done purposely to assure that students were motivated.

3.1 Motivation for Conducting the Replication

The main reason for conducting a replication was to confirm the results of the original experiment or, in the event of inconsistencies, identify the factors and parameters that might have caused such inconsistencies. This could, if necessary, trigger another replication cycle. The independent experiment replication should, secondarily, help to improve our competence at applying empirical methods in SE research.

3.2 Level of Interaction with the Original Experimenters

There was a lot of interaction with the original experimenters in the phases prior to the execution of the experimental replication. The replication was executed without the involvement of the original experimenters. Finally, the original experimenters played a very minor role in data collection and analysis.

(A) At the start of the replication we had to gain a profound understanding of the original experiment. For this purpose we held several meetings with

the original experimenters. Once we had a thorough understanding of the experiment, the next step was to adapt the design of the original experiment to the context where the replication was to be carried out. As a result of this adaptation, we generated a design document, which was validated by the original experimenters. Finally, we were given the artefacts of the original experiment.

(B) We did not have any interaction with the original experimenters during the preparation of the training materials, training proper and replication execution.

(C) After we had executed the replication, the original experimenters explained the procedure for identifying whether or not a test case reveals a fault from the questionnaires completed by each experimental subject.

3.3 Changes to the Original Experiment

Generally speaking, the replication is quite true to the original experiment in all respects. The hypotheses, factors, faults, response variable, materials and experimental procedure are all unchanged, save the following exceptions:

- We have eliminated one of the main factor levels. Specifically, we did not test the code reading technique, primarily because it was impossible to run three experimental sessions in the time available for running the replication.
- As we have omitted one of the techniques, one of the three sessions and one of the three programs are unnecessary. In fact, we have omitted the cmdline program and reduced the number of sessions from three to two.
- We have altered the order in which the programs were used in order to study whether it is the program or the session that influences technique effectiveness.
- We have run the replication with a group of experimental subjects that have different characteristics than the subjects of the original experiment.
- We have applied stratified randomization (Kernan et al. 1999) to assign subjects to experimental groups in order to assure balanced groups.
- We have adapted the training to the time constraints of the course on which the replication was run.
- We have localized the experimental materials to the dialectal differences of Ecuadorian Spanish.

Table 4 summarizes the changes. The changes are described in more detail in the following.

3.3.1 Changes to the Main Factor Levels

According to the software verification and validation course schedule, only two days were available for running the experiment, whereas three days (one per session) had been spent on the original experiment. We had two options in this respect:

1. Squeeze two experimental sessions into one day.
2. Eliminate one of the technique factor levels.

A third option, which meant splitting a session across two days, was rejected outright as there was a risk of the intermission affecting student performance or simply of students swapping notes.

Table 4 Differences between UPM and ESPEL

Activity	Characteristic	UPM	ESPEL
Design	Randomization	Normal	Stratified
	Sessions	3	2
	Main factor	CR, BT, EP	BT, EP
	Programs	cmdline, ntree, nametbl	nametbl, ntree
	Groups	6	2
	Dialectal differences in materials	Castilian Spanish	Ecuadorian Spanish
Recruitment	Number	46	23
	Type	Undergraduate students	Master's students
	Professional experience	Generally inexperienced	Yes
Training	Training type	Face-to-face	Semi-distance
	Duration	12 h	50 h
Execution	Duration	3 sessions, unlimited time	2 sessions, unlimited time

Finally, we went for the second of the two options. The fact that the sessions were scheduled for two consecutive days (Saturday and Sunday) went against the first option: it would have been very demanding on students to take part in three sessions without a break, and the resulting fatigue could have had a negative influence on the results. Also, the fact that the original experiment really tested the functional and structural techniques, whereas code reading was primarily a control technique, and therefore optional, favoured the second option.

Consequently, the factor levels were the functional testing technique known as *equivalence partitioning* (EP) and the control-flow structural testing technique known as *branch testing* (BT), each applied in one session by two groups of experimental subjects (Group 1 and Group 2). Table 5 shows the resulting experimental design.

3.3.2 Changes to the Secondary Factor Levels

The omission of one the testing techniques (code reading) makes one of the three sessions unnecessary. Consequently, it was also necessary to eliminate one of the programs used in the original experiment. We left out the cmdline program for two key reasons:

1. Experimental subjects stated, during the original experiment and other experiments of the same family (Juristo and Vegas 2003), that the *cmdline* program was a harder to understand and test than *ntree* and *nametbl*. Testing techniques are generally less effective on cmdline, suggesting that this program is more complex, as discussed in Section 2.7.
2. The ntree and nametbl programs are similar to each other (146-172 LOC and a cyclomatic complexity of 21-29, respectively), and both are different to

Table 5 Replication design

Session	Session 1		Session 2	
Program	Nametbl		Ntree	
Technique	BT	EP	BT	EP
Group 1	X	–	–	X
Group 2	–	X	X	–

cmdline (209 LOC and a cyclomatic complexity of 61). cmdline's high cyclomatic complexity is a possible explanation for it being harder to test.

In the original experiment (see Table 1), the cmdline program was used in Session 1, whereas ntree and nametbl were used in Sessions 2 and 3, respectively. This would apparently signify that the sessions in the original experiment and the replication are not comparable, as they are associated with different programs. However, this should not be a problem as there is unlikely to be any relationship between session and program, as already stated.

3.3.3 Change to the Order of Program Use

As discussed in Section 2.7, the analysis is unable to distinguish whether the effects are due to either factor because program/session are confounded. This is a peculiarity of the cross-over design used in the original experiment. In this replication, they are again confounded because we adhere to the original design. On this ground, we have decided to change the order in which the programs are used from ntree then nametbl in the original experiment to nametbl then ntree in the replication. By comparing the original and replicated experiment, we expect to be able to identify whether the possible effects are really due to the program or the session.

3.3.4 Change to Population Type

The subjects of the original experiment at the UPM were undergraduate students, most of whom had little or no professional experience. On the other hand, the subjects of the replication at ESPEL were master's students, many of whom did have professional programming, architectural design or other software development experience.

Because of their higher academic and professional level, the marginal means of ESPEL subjects should be greater than for UPM subjects. However, this should not affect the comparisons between factors and treatments, as the implemented stratified randomization (see Section 3.3.5) assures that the experimental groups are homogeneous.

3.3.5 Balancing Experimental Groups

The design of the replication called for the formation of two groups of experimental subjects called Group 1 and Group 2. These two groups were formed from 23 experimental subjects. Because the subjects of the ESPEL replication have a different level of professional experience than the subjects of the original experiment, we decided to stratify the groups depending on this characteristic to assure a more reliable balancing of groups. To do this, we conducted a survey of 21 subjects (two did not attend on the day that the survey was administered) to get a better picture of their professional experience, assuming that professional programmers or testers would apply the techniques more effectively than subjects with less experience.

In the survey, the experimental subjects were asked about their general programming experience, C programming experience and software testing technique experience. The results are illustrated in Figs. 1, 2 and 3, and tabulated in Table 21 of Appendix B.

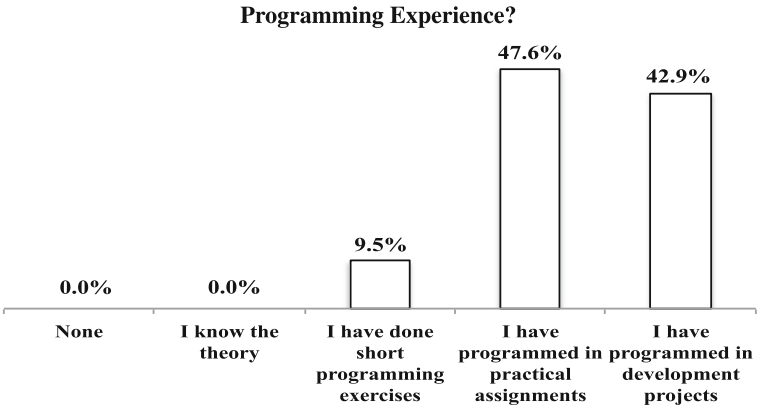

Fig. 1 Survey results—programming

We found that almost half of the experimental subjects are professional programmers, whereas the other half have only programmed as part of practical assignments. C is not the most popular programming language used by subjects for formal development, but they are all acquainted with the language, at least in theory. A considerable percentage (20 %) of experimental subjects are not familiar with software testing techniques. None are professional testers, and generally they have only used testing techniques as part of practical assignments.

Experience in both C and the testing will definitely have an effect. However, 81 % of subjects have no professional experience in C, whereas none of the subjects have professional testing experience. The difference in the effectiveness between inexperienced and experienced subjects will be very small for both variables, as the value range is from *No experience* to *I have done practical assignments*. Therefore, we can assume that experience in C and testing will have a rather small effect or, at least, less than professional experience in programming is likely have.

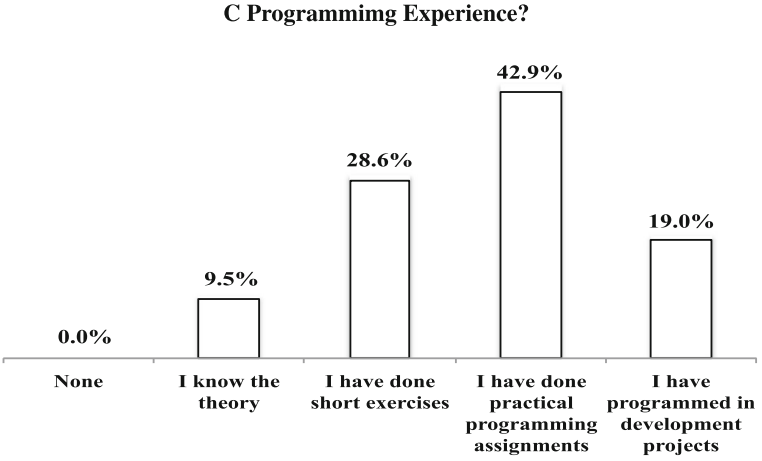

Fig. 2 Survey results—C programming

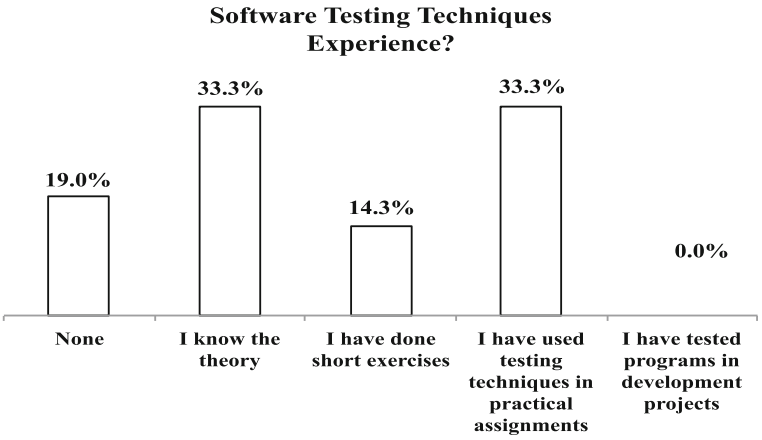

Fig. 3 Survey results—software testing

On the other hand, programming experience did vary considerably (roughly 40 % of subjects have professional programming experience, compared with 60 % who do not), and it is reasonable to assume that this experience may well have a bearing. On this ground, we conducted a stratified randomization (Kernan et al. 1999) based on programming experience. The two subjects that were not surveyed were each allocated to one group (G1 (EP-BT) or G2 (BT-EP)) at random.

In this way, we produced two groups of experimental subjects that were balanced with respect to programming knowledge. With respect to C programming experience, both groups were balanced regarding the number of subjects that used C in industry or academia (around 20 and 80 %, respectively), as shown in Fig. 4. G1 subjects had a slight advantage in terms of the type of experience that they had acquired in academia. Of G1 group subjects, 54 % have used C in practical assignments, whereas the remaining 27.3 % know the theory or have completed short exercises. The respective percentages for the G2 group are 30 and 50 %.

The G1 group also appears to have slightly more software testing technique experience, as shown in Fig. 5. Of these subjects, 45.5 % have experience in practical assignments, whereas the remaining 54.6 % know the theory or have completed short exercises. The respective percentages in the G2 group are 20 % and 80 %.

This imbalance between groups could result in subjects from the EP-BT group being generally more effective than subjects from the BT-EP group. Although we think this is a remote possibility, it should be taken into account during the discussion of the results of the replication. The between-group differences are confined to some subjects in the G1 group having completed more practical assignments than G2 group members. We have the feeling that any difference there is will be small. Additionally, all subjects have received special-purpose training in software testing before the experiment, which includes practical exercises. This training should have further reduced the differences between the groups.

3.3.6 Training Adaptation

The teaching method applied in the original experiment was divided into three four-hour sessions plus homework, each of which was held one week apart. The

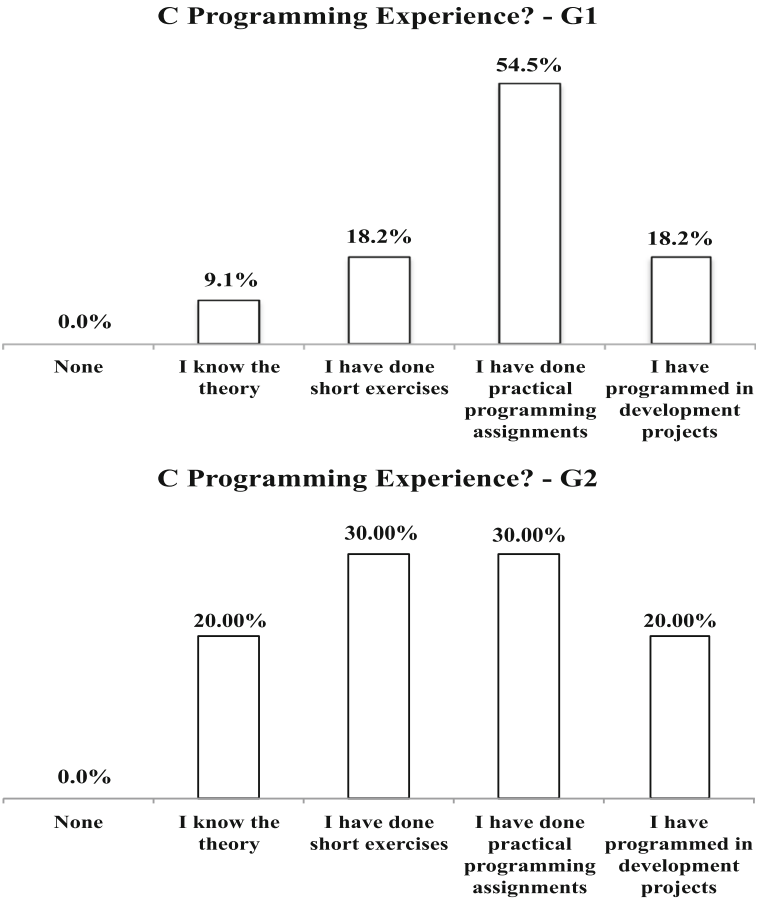

Fig. 4 Results of the stratified randomization for C programming

replication training, on the other hand, had to be adapted to the semi-distance teaching method at ESPEL, with five consecutive 10-h sessions, during which students completed all the practical exercises. In view of this training method, the fatigue factor is a validity threat, which may have influenced subject performance.

3.3.7 Localization

Although similar syntactically, the Spanish spoken in Spain (Castilian) and the Spanish spoken in Latin America, particularly Ecuador, are slightly different with respect to the words and idioms used locally. On this ground, we modified details, such as terms and phrases, that are not common in the dialect spoken in the area where the replication was conducted to ease understanding. We also corrected some minor ambiguities found in the original material. The material used for replication purposes is available at: http://www.grise.upm.es/sites/extras/12/.

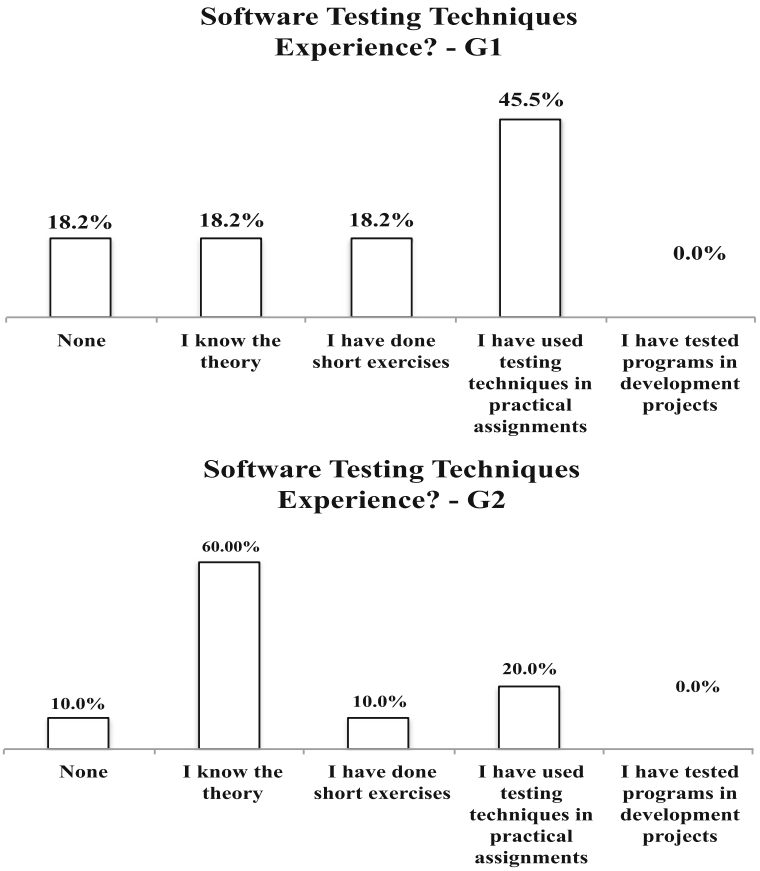

Fig. 5 Results stratified randomization for testing experience

3.4 Replication Execution

The replication was executed according to a similar procedure to the original experiment. The subjects were given the same experimental materials (experimental objects, program specifications, source code listing with seeded faults, forms, guides and executable code with seeded faults), and generated and proceeded to execute test cases in order to check that the identified faults caused failures. There were no major events (e.g. drop-outs, errors in materials delivery, etc.) during the replication execution. The data were analyzed according to the same procedure as in the original experiment.

Table 6 Test of within-subjects effects

Source	Type III sum of squares	df	Mean square	F	Sig.
Technique	1999.054	1	1999.054	4.517	0.046
Session/program	4655.948	1	4655.948	10.520	0.004
Error (technique)	9294.241	21	442.583		

Table 7 Test of between-subjects effects

Source	Type III sum of squares	df	Mean square	F	Sig.
Intercept	68201.322	1	68201.322	100.846	0.000
Group	6028.757	1	6028.757	8.914	0.007
Error	14202.195	21	676.295		

3.5 Replication Results

This section describes the results of the replication. Specifically, we report the hypothesis tests and multiple comparisons for each response variable (InScope or OutScope). The raw data and descriptive statistics are reported in Appendix C.

Like the original experiment, the experimental design used in the replication lends itself to a repeated-measures analysis of variance (rANOVA) and the same additive model. SPSS V.20 was used for all calculations.

3.5.1 Response Variable: Effectiveness for Faults Within Technique Scope

There are two requirements for applying a rANOVA: homogeneity of the covariance matrices and sphericity.

Box's M test is used to check the condition of homogeneity of covariance matrices. Box's M tests the null hypothesis that the observed covariance matrices of the dependent variables are equal across groups (Meyers et al. 2006). For our sample, M = 3.755, F = 1.122, df1 = 3, df2 = 111064.484, sig. = 0.338, that is, the results verify the null hypothesis and the data are, therefore, homogeneous.

Mauchly's test is used to check the sphericity condition. In our case, however, there are only two levels of repeated measures (for both technique and session/program), which precludes a sphericity violation (Meyers et al. 2006), and, therefore, the test is unnecessary.

As the analysis contains within- and between- subjects factors, we obtain two different tables, one for each factor type, instead of the standard ANOVA table (Tables 6 and 7). The results suggest that the technique, session/program and group factors all influence the effectiveness with respect to the detection of faults within technique scope. Therefore, the null hypothesis (there is no difference in the effectiveness of *equivalence partitioning* and *branch testing* with respect to the detection of faults within their scope) for this response variable is rejected.

The results of the pairwise comparison for the **Technique** factor, which are shown in Table 8, suggest that *branch testing* is less effective than *equivalence partitioning* (with an effectiveness of 31.943 and 45.140 % respectively).

The pairwise comparisons for the **session/program** factor suggest that **S1/nametbl** is more effective than **S2/ntree** (with an effectiveness of 48.612 and 28.471 %, respectively), as shown in Table 9. As only one program is used in a session, it is not possible at this point to distinguish whether the effect is produced by the

Table 8 Pairwise comparisons test for technique

Tech1	Tech2	Mean dif.	Std. dev.	Sig.
BT	EP	−13.197	6.210	0.046

Table 9 Pairwise comparisons test for program

Prog1	Prog2	Mean dif.	Std. dev.	Sig.
S1/nametbl	S2/ntree	20.140	6.210	0.004

Table 10 Pairwise comparisons test for group

Group1	Group2	Mean dif.	Std. dev.	Sig.
BT-EP	EP-BT	−22.918	7.676	0.007

Table 11 Test of within-subjects effects

Source	Type III sum of squares	df	Mean square	F	Sig.
Technique	2905.36	1	2905.36	5.567	0.028
Session/program	6.577	1	6.577	0.013	0.912
Error (technique)	10959.95	21	521.902		

Table 12 Test of between-subjects effects

Source	Type III sum of squares	df	Mean square	F	Sig.
Intercept	17357.588	1	17357.588	20.086	0.000
Group	354.129	1	354.129	0.41	0.529
Error	18147.296	21	864.157		

Table 13 Pairwise comparisons test for technique

Tech1	Tech2	Mean dif.	Std. dev.	Sig.
BT	EP	15.910	6.743	0.028

Table 14 Pairwise comparisons test for program

Prog1	Prog2	Mean dif.	Std. dev.	Sig.
S1/nametbl	S2/ntree	0.757	6.743	0.912

Table 15 Pairwise comparisons test for group

Group1	Group2	Mean dif.	Std. dev.	Sig.
BT-EP	EP-BT	5.554	8.677	0.529

session, by the program or by both. We will try to clarify whether the session or the program caused the observed effect when we examine the OutScope faults and, especially, when we compare the replication results with the original experiment.

The results of the pairwise comparison for the **Group** factor suggest that the BT-EP group (which applies *branch testing* in session 1 followed by *equivalence partitioning* in session 2) is less effective than the EP-BT group (with an effectiveness of 27.082 and 50.00 %, respectively), as shown in Table 10.

There are two possible explanations for this result:

- A carryover effect could be influencing the effectiveness of techniques depending on whether they are applied in first or second place. Carryover signifies an increase (or decrease) in the effectiveness of the technique that a subject applies in second place.
- As mentioned earlier in Section 3.3.5, the EP-BT group is slightly more experienced than the BT-EP group in C and software testing. This superior experience could explain why the EP-BT group is more effective.

We will try to determine whether the observed effect is to due to between-group differences or carryover when we study the OutScope faults and, later, when we compare the results of the replication with the original experiment.

3.5.2 Response Variable: Effectiveness for Faults Outside Technique Scope

We check whether the sample has the sphericity and homoscedasticity properties before conducting the statistical analysis. Mauchly's W and Box's M statistics again confirmed those properties ($W = 1.000$, Approx. Chi-Square $= 0.000$, $df = 0$, Sig. $= 0.000$; $M = 11.947$, $F = 0.429$, $df1 = 3$, $df2 = 111064.484$, Sig. $= 0.732$).

The analysis results, which are shown in Tables 11 and 12, suggest that there are significant differences for the technique factor, but not so for the session/program and group factors. Therefore, as in the case of the InScope response variable, the null hypothesis is rejected.

Regarding the **Technique** factor, Table 13 shows that *branch testing* is more effective than *equivalence partitioning* for faults outside technique scope (with an effectiveness of 31.943 and 11.489 %, respectively). These results reveal the opposite pattern to the analysis of the InScope faults reported in Section 3.5.1 (that is, *branch testing* is less effective than *equivalence partitioning* for faults within technique scope). This suggests that *equivalence partitioning* is more sensitive to faults within its scope, but *branch testing* is better at detecting faults outside its scope.

The multiple comparisons for the **session/program** factor, shown in Table 14, suggest that there are no significant differences between the levels of this factor. However, the results of the InScope response variable, which were reported in Section 3.5.1, did suggest that there were significant differences. There is no apparent reason why the session/program factor should behave differently depending on fault types. Therefore, we are unable to venture any hypothesis to explain this discrepancy considering just the replication results.

The multiple comparisons for the **Group** factor, shown in Table 15, suggest that, unlike our findings for the InScope response variable, there are no significant differences between BT-EP and EP-BT.

In Section 3.5.1, we ventured two hypotheses to explain the results for InScope faults: the existence of a carryover effect or, alternatively, the EP-BT group's supe-

rior C and testing technique experience. In either case, we would expect this effect to be the same irrespective of the fault type. Again considering only the replication results, we are unable to venture any reason why effects should be significant for the InScope response variable.

4 Comparison of Replication Results to Original Results

Because our replication uses a subset of factor levels of the original experiment, we will not be able to contrast all the results of the original experiment with the replication, and some will be only partially comparable. The *branch testing* and *equivalence partitioning* levels of the technique factor are comparable in both experiments, but we left out the *stepwise abstraction* technique, which is used only in the original experiment.

The session/program factor is partially comparable, as the programs used in the two experiments (nametbl and ntree) correspond to different sessions in each experiment (sessions 1 and 2 in ESPEL and sessions 3 and 2 in UPM, respectively). We clearly define whether we are referring to the program or session in each case.

In the case of the group factor, it is the technique factor levels and experiment sessions that define the levels of each group. Six different groups were formed in the original experiment and only two in the replication. The two groups formed in the replication are subsets of two of the six groups of the original experiment. Therefore, they are only comparable at a very high level.

Sections 4.1 and 4.2 contain the similarities and differences, respectively, between the original experiment and the replication.

4.1 Consistent Results

The consistent results between the original experiment and the replication refer to the technique factor for both the InScope and OutScope response variables, as shown in Table 16. This table summarizes the results obtained in the multiple comparisons for the original experiment (UPM) and the replication (ESPEL), specifying whether the null hypothesis (there is no difference in the effectiveness of *equivalence partitioning* and *branch testing* with respect to the detection of faults) is rejected or accepted and showing the observed pattern (order relationship) between the factor levels.

4.1.1 InScope Response Variable

Branch testing is less effective than *equivalence partitioning* in both experiments, albeit with some fine distinctions. Firstly, as shown in Table 16, at ESPEL we have obtained significant differences between the technique levels, whereas the difference was not so grand at UPM. However, the patterns are the same (BT < EP) in both cases.

Table 16 UPM-ESPEL comparison—technique factor

Response	H0		Tendency	
Variable	Upm	Espel	Upm	Espel
InScope	Accept	Reject	$BT < EP$	$BT < EP$
OutScope	Reject	Reject	$BT > EP$	$BT > EP$

Secondly, the marginal means for the technique factor at ESPEL are lower than at UPM (*branch testing* with 31.943 vs. 67.670 % and *equivalence partitioning* with 45.140 and 78.704 %, respectively). Finally, *branch testing* technique dispersion is lower at ESPEL than at UPM, as shown in Fig. 6.

Branch testing's lower dispersion at ESPEL may explain why the difference between *branch testing* and *equivalence partitioning* is significant at ESPEL. BT's wider dispersion at UPM may be due to the fact that UPM subjects are undergraduate students, in general without professional experience, and therefore their testing abilities may vary considerably. This would generate wide interquartile ranges. As a consequence, the null hypothesis could not be rejected at UPM, causing the

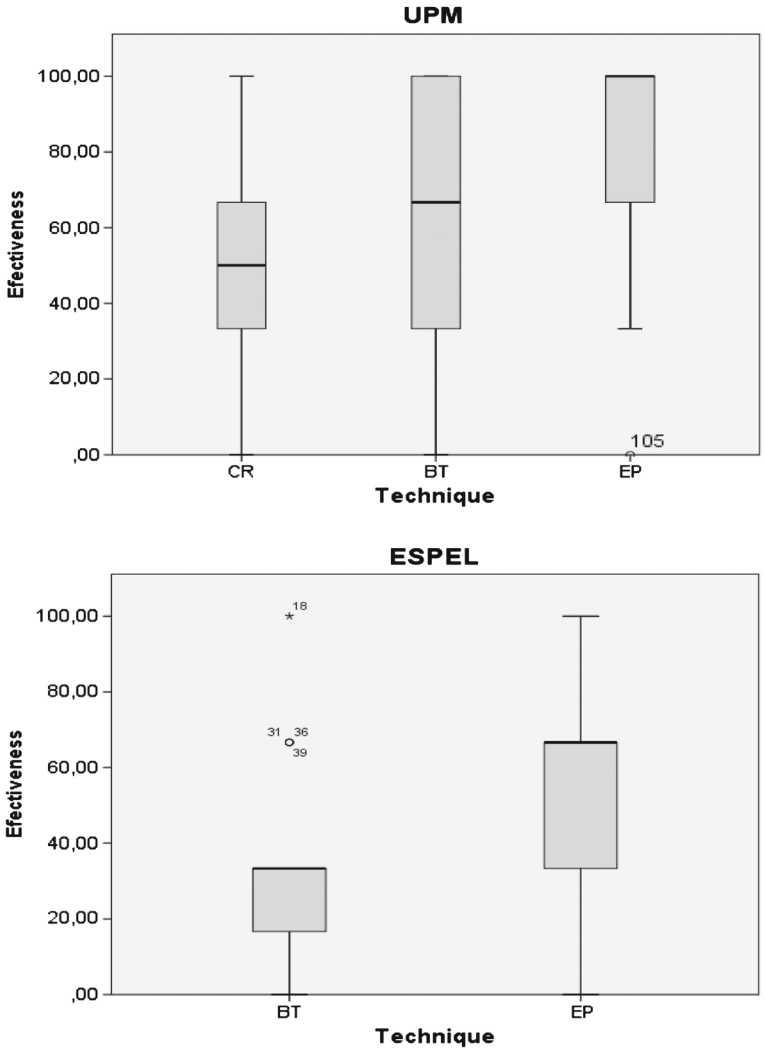

Fig. 6 Boxplot for technique at UPM and ESPEL—InScope

impression that the results at UPM and ESPEL are slightly different, when they really are consistent.

Why the subjects are less effective at ESPEL than at UPM is another question. The most likely reason is that the course on which the experiment was run is intensive (long lecture hours concentrated over just a few days). This may have influenced technique effectiveness, as subjects may not have had enough time to practice and consolidate the usage of the techniques. *Branch testing* would be more adversely affected, since subjects told us at post-experimental meetings that BT is harder to understand and apply than EP. Another potential factor, besides the teaching method, is trainer inexperience in teaching the software verification and validation course, especially under such circumstances.

4.1.2 OutScope Response Variable

The results obtained at UPM for this response variable were confirmed at ESPEL, as the null hypothesis is rejected in both cases, and, besides, the trend is the same as shown in Table 16, where *branch testing* is more effective than *equivalence partitioning* for faults that are outside their scope. The values of the marginal means are slightly lower at ESPEL (27.398 % for *branch testing* and 11.489 % for *equivalence partitioning*) than at UPM (29.089 % for *branch testing* and 14.119 % for *equivalence partitioning*). The dispersions are generally quite similar, albeit, predictably for undergraduate students, slightly wider at UPM, as shown in Fig. 7. It is interesting to note that the low technique effectiveness at ESPEL is much more marked for the InScope than for the OutScope faults, considering that the differences in the training (both course intensiveness and possibly trainer inexperience) should (in principle) affect both response variables more or less equally.

A possible explanation for better *branch testing* performance with OutScope faults is that the test case generation strategy requires an analysis of source code, at which point subjects could informally apply code inspection and thus round out the technique. This is a convincing explanation for two reasons:

- Subjects applying the *equivalence partitioning* technique do not have the source code of the program that they are testing (only the executable), as mentioned in Section 2.6.2. Consequently, their fault detection proficiency should be very low. This is precisely what we found, as the mean effectiveness of experimental subjects is 11.5 %, that is, each subject identifies on average 0.3 faults.
- Without the help of testing techniques, subjects with comparable characteristics (programming experience, years of experience industry, etc.) should locate more or less the same faults. We expect to observe that ESPEL subjects are more effective than UPM subjects because they have some professional experience. We found that the effectiveness of both subject groups (ESPEL and UPM) is more or less equal. Now, this is precisely what we would expect to find if the ESPEL subjects were to be suffering from fatigue as a result of experiment planning, as discussed in Section 4.1.1. Another reasonable hypothesis in the light of the results is that subjects apply informal code reading during the application of the *branch testing* technique.

Even though this is a reasonable hypothesis, it needs to be corroborated in future replications.

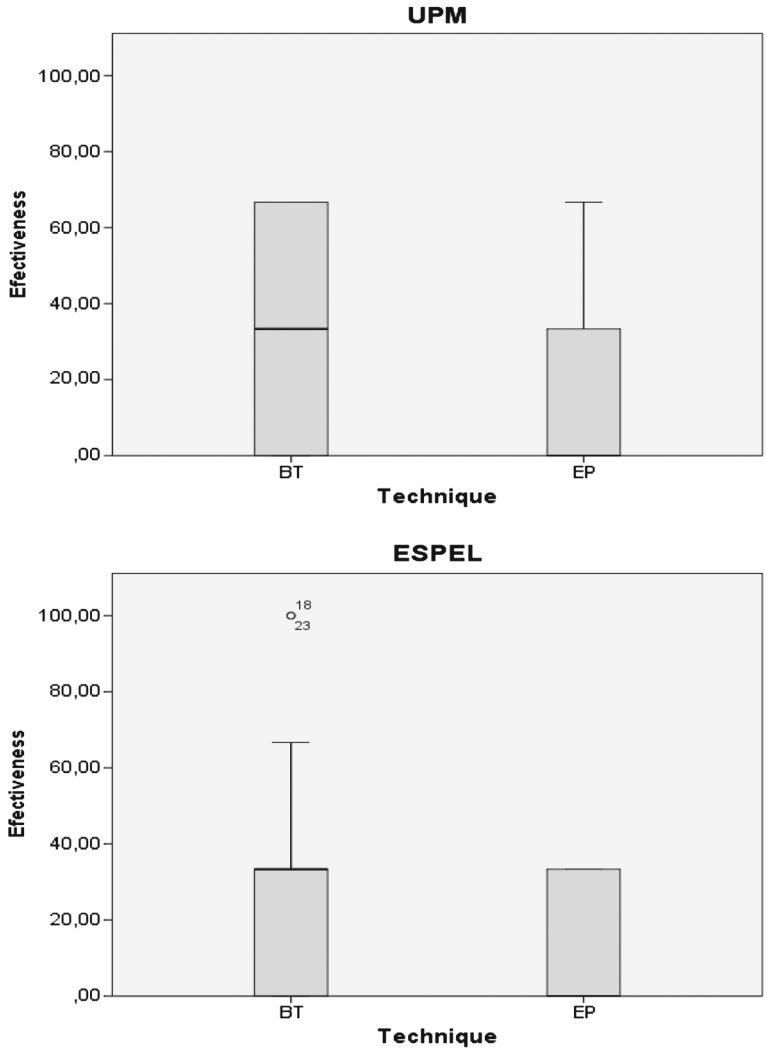

Fig. 7 Boxplot for technique at UPM and ESPEL—OutScope

4.2 Differences in Results

The differences between the results of the original experiment and the replication refer to the session/program and group factors. Both are dealt with separately in the following.

4.2.1 Differences with Respect to the Session/Program Factor

Table 17 summarizes the comparison for the session/program factor.

(A) **InScope Response Variable** With respect to **session/program** factor, testing was more effective at UPM when the ntree program was applied, whereas just

Table 17 UPM-ESPEL comparison—session/program factor

Response	H0		Tendency	
Variable	Upm	Espel	Upm	Espel
InScope	Accept	Reject	$S3 < S2$	$S1 > S2$
			$ntbl < ntree$	$ntbl > ntree$
OutScope	Accept	Accept	$S3 > S2$	$S1 < S2$
			$ntbl > ntree$	$ntbl < ntree$

the opposite was the case at ESPEL, where nametbl was more effective as shown in Fig. 8. Additionally, the differences are significant at ESPEL, whereas at UPM they are not.

We changed the order in which the ntree and nametbl programs were applied in the replication. This change is designed to clarify whether it is the session or program (as the original experimenters claim) that is causing the observed effect.

ESPEL results contradict UPM results, suggesting that causal factor is probably the session and not the program.

Fatigue could be the mechanism through which the session influences effectiveness. Note that the experiment was run on an intensive academic programme and experimental sessions were held only one day apart. Therefore, subjects might well have been fatigued when they arrived at the experimental sessions (notice, in this respect, that the marginal means of effectiveness are always lower at ESPEL than at UPM and that, in particular, S2 was less effective than S1 (as Fig. 8 clearly shows).

The possibility of a fatigue effect is a convincing hypothesis on two grounds. Firstly, the sharp drop in effectiveness from S1 to S2 would explain why the S1/nametbl and S2/ntree differences turned out to be significant. If fatigue had had no effect, the differences would have been smaller and possibly not significant, which is what was found at UPM (where the sessions were held one week apart). Secondly, the effect size for the session/program factor (shown in Table 9) is abnormally high compared with the technique factor (shown in Table 8). The fatigue effect is also compatible with this finding.

In any case, the data gathered in the two experiments are still not enough to determine which factor (session or program) is really having a bearing, so yet more replications will be necessary to clarify this point.

(B) **OutScope Response Variable** Neither experiment observed significant differences with respect to the OutScope variable. The above-mentioned possibility of a fatigue effect is entirely consistent with the fact that S2/ntree is less effective than S1/nametbl, as shown in Fig. 9.

The ESPEL and UPM results have a completely different pattern, giving the visual impression that the two experiments are inconsistent. Note, however, that the S1–S2 and ntree-nametbl differences are not significant in either experiment. The fact that the results are not significant shows that neither the program nor the session (with the possible exception of a fatigue effect) has any effect on OutScope fault detection effectiveness, which makes sense. This strengthens the plausibility of OutScope fault detection depending exclusively on the expertise of the experimental subjects, as specified in Section 4.1.2.

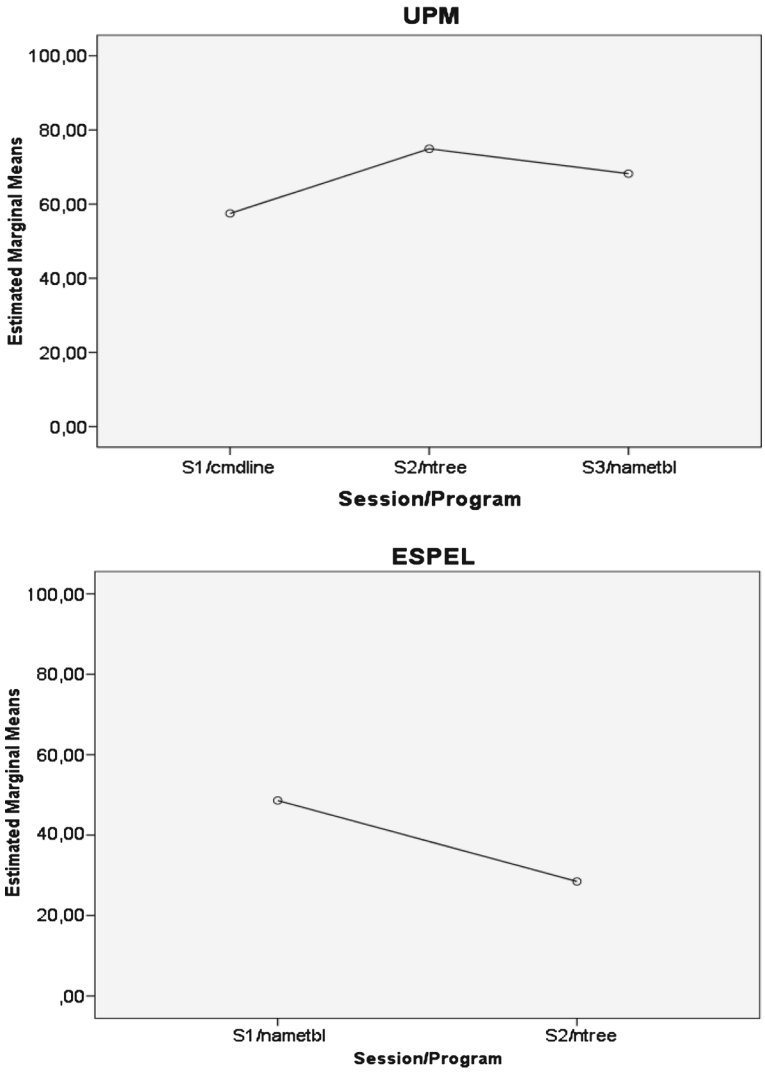

Fig. 8 Estimated marginals means for Session/Program at UPM and ESPEL—InScope

4.2.2 Differences Regarding the Group Factor

The between-group differences at ESPEL and UPM cannot be tabulated and plotted as above, because they are too profound. Taking into account the statistical significance of the results, however, we can compare the two experiments as shown in Table 18.

(A) **InScope Response Variable** The results of the experiments at both UPM and ESPEL are statistically significant for the group factor. The analysis suggests that ESPEL results may be due to either a carryover effect or an imbalance in experience in C programming and testing technique use across experimental

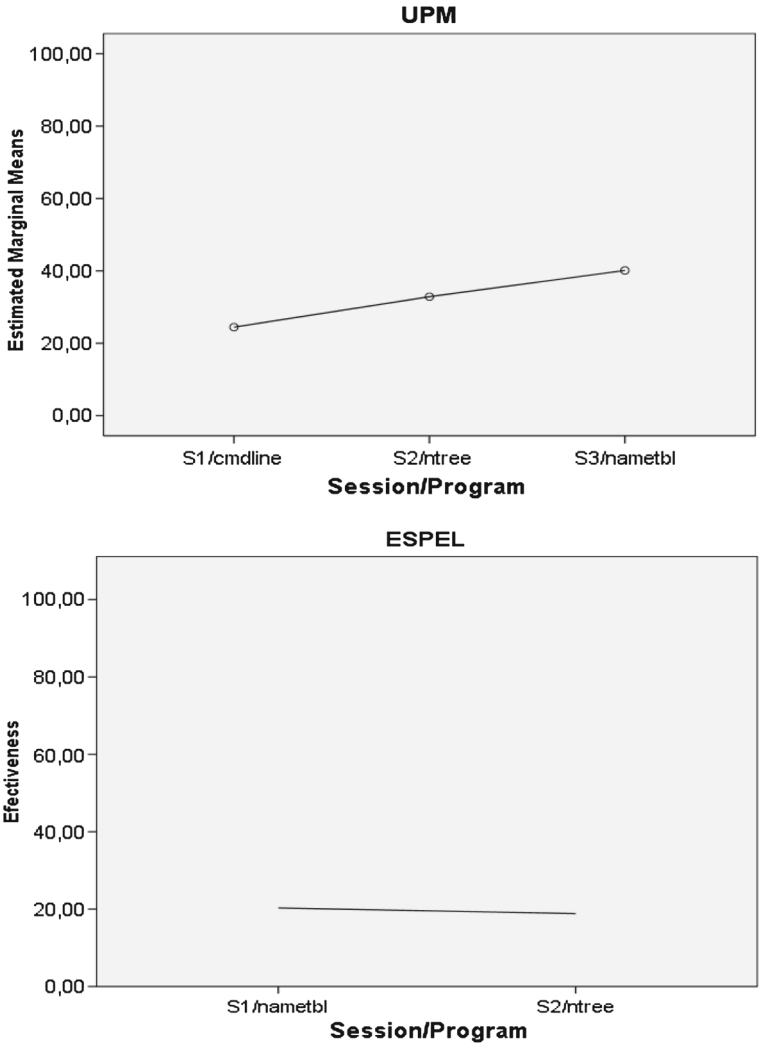

Fig. 9 Estimated marginals means for session/program at UPM and ESPEL—OutScope

groups. At UPM, we do not know whether or not the experimental groups are balanced (the original experimenters do not provide this information). Additionally, the original report (Juristo et al. 2013) indicates that there was no carryover.

Table 18 UPM-ESPEL comparison—group factor

Response	H0	
Variable	Upm	Espel
InScope	Reject	Reject
OutScope	Reject	Accept

We take the view that the between-group imbalance has not had a decisive impact. The imbalance between experimental groups at ESPEL should show up consistently across the InScope and OutScope faults. However, this is not the result that we observed, as the between-group difference for OutScope faults is not significant (and, besides, is different to the pattern for InScope faults).

In our view, the carryover effect cannot be ruled out. This opinion is based on two observations:

- The analysis of the technique and session/program factors at both ESPEL and UPM appears to suggest that OutScope fault detection depends exclusively on the experimental subjects, that is, on their knowledge, experience, etc. If this were the case, we should not observe any carryover effect for OutScope faults in the replication, as *equivalence partitioning* and *branch testing* have no influence on OutScope fault detection. This would not apply to InScope faults, and it would make sense if we were to observe a carryover effect.
- Nonstatistically, using EP first appears to improve the effectiveness of the techniques applied subsequently in both experiments (groups EP-CR-BT and EP-BT-CR in the original experiment and group EP-BT in the replication appear to be more effective).

Small sample effects offer a possible explanation, which does not support the carryover effect, for the significant differences that show up with respect to the group factor. The original experiment had 46 experimental subjects, so the number of subjects per group is $46/6 \simeq 8$. At ESPEL, the number of subjects per group is $23/2 \simeq 12$. With so few subjects per group, the significant effects may be a product of chance.

We take the view that the small sample effects can add noise to the data analysis (that is, cause some groups to exhibit significant differences from others purely by chance), but this does not fully explain the results. Note that the small sample effect should act consistently across InScope and OutScope faults, which is not the case. At UPM, three groups exhibit significant differences with respect to InScope faults, whereas only one of the groups has significant differences with respect to OutScope faults (which could quite possibly be a small sample effect). At ESPEL, the differences are significant for InScope but not for OutScope faults. Consequently, there may well be a carryover effect with respect to InScope faults.

Unfortunately, the carryover hypothesis is rather speculative. The groups at ESPEL and UPM are not directly compatible; hence all inferences are based on indirect evidence. Consequently, more replications need to be conducted to confirm or reject the existence of a carryover effect.

(B) **OutScope Response Variable** We have been obliged to explain all our findings with respect to the OutScope variable in the analysis of the InScope response variable. On this ground, we will merely state our conclusions at this point:

- We ascribe the existence of significant differences with respect to the group factor for OutScope faults to a small sample effect.
- It appears from the analysis of the technique and session/program factors that subjects draw on their own expertise to detect OutScope faults. On

this ground, there should be no carryover effect (which, basically, is the end result of a relationship between testing techniques and cannot, therefore, exist unless they have a bearing on the OutScope response variable).

5 Conclusions and Lessons Learned Across Studies

5.1 Conclusions

Comparing non-identical replications is a complex issue. The changes caused by eliminating one of the technique factor levels on logistic grounds (insufficient time to run all the experimental sessions) have had a waterfall effect on the program and session factors. Consequently, neither the sessions nor the groups are comparable in every respect. Even so, the replication has helped to get a better understanding of the influence of the factors under study:

- Firstly, we have confirmed that the *equivalence partitioning* technique is more effective at detecting faults that are within its scope and *branch testing* is more effective for faults outside its scope. A possible reason for this difference of effectiveness in the case of the InScope variable is that subjects find the *branch testing* technique harder to use or, at least, are better at applying the *equivalence partitioning* technique. Regarding the difference in effectiveness for the OutScope response variable, we hypothesize that the structural technique is more effective because students use code review to round out the technique. As they have access to the source code (not so for the functional technique), they can inspect the code to gain a better understanding of the program. The statistical results show that the measures of effectiveness were generally lower at ESPEL than at UPM. This could be attributed to the influence of the setting. To be precise, we believe that the extremely intensive teaching method applied in training could have had an influence. Apart from the teaching method, another consideration is trainer inexperience in teaching the software verification and validation course, especially under the circumstances.
- Secondly, the results for the session/program and group factors with respect to the OutScope response variable support the hypothesis that neither the EP nor the BT technique really influence OutScope fault detection. In both cases (session/program and group), the results were not significant in either the original experiment or the replication.
- Thirdly, the existence of some sort of carryover effect for the InScope variable appears to be confirmed. In this case, the problem is that the groups are formed differently in the two experiments and are hence not comparable. Consequently, further replications need to be conducted before we can state that this effect really does exist.

The hardest thing to figure out was the influence of the session/program factor with respect to the InScope variable. The original experimenters had concluded that the program was responsible for the session/program effect on effectiveness at UPM (remember that the two factors are confounded). The original experimenters arrived at this conclusion after comparing the InScope and OutScope fault detection effectiveness for session/program. This comparison suggested that the differences

between cmdline, nametbl and ntree together offered a better explanation than the differences across sessions S1, S2 and S3 for the resulting data.

However, the results at ESPEL for InScope have a completely opposite pattern to UPM findings. Not only do the ESPEL and UPM patterns differ with respect to the programs (ntree>nametbl at UPM, whereas nametbl>ntree at ESPEL), but the differences at ESPEL are also statistically significant.

One of the changes made to the replication with respect to the original experiment was to reduce the number of sessions from three to two. This led to one of the programs used in the original experiment (cmdline) being omitted. Under these circumstances, it is hard to reach any sort of reliable conclusion. Our analysis tends to ascribe the observed effects to the session rather than to the program. However, there are other alternative explanations. The existence of some sort technique/program interaction is a particularly convincing cause.

Figure 10 shows boxplots for the technique factor. In contrast to Fig. 6, they have been further decomposed by session/program. Figure 10 is not easy to interpret because there are several outliers, but the median for EP x nametbl is clearly much greater than for the other combinations (BT x nametbl, etc.). The profile diagram shown in Fig. 11 illustrates these values more clearly (Note that this diagram plots means not medians). Neither the original experiment nor the replication (which reuses the original analytical model for the sake of comparability) account for this interaction, as a repeated-measures ANOVA cannot calculate this effect.

Irrespective of the analytical model used, the existence of such an interaction would be compatible with the results for technique and program at ESPEL and would explain why a carryover effect was observed. (Note that the technique/program interaction, that is, where a technique may be more effective when applied to certain programs, is confounded with the group factor, and thus the statistical analysis is unable to distinguish the two effect types.) A possible technique/program effect does not explain all the findings, however. In particular, there is the question of

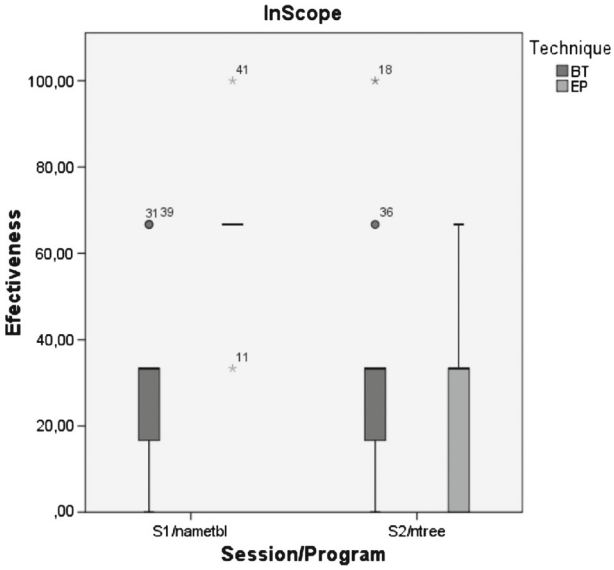

Fig. 10 Boxplot for the interaction technique × session/program at ESPEL—InScope

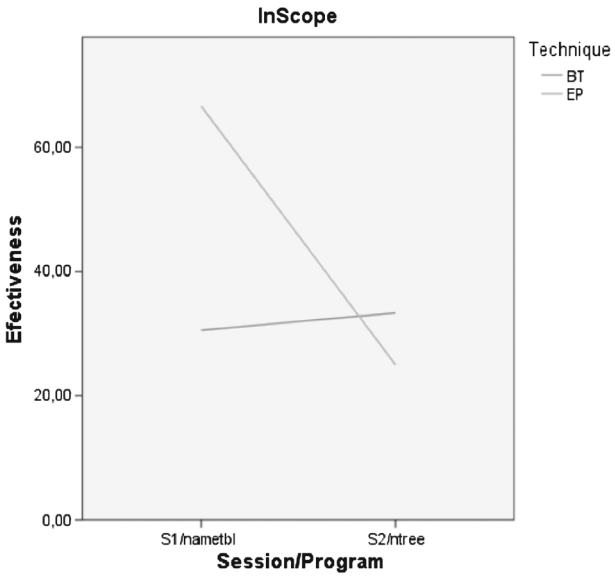

Fig. 11 Profile diagram for the technique × session/program interaction at ESPEL—InScope

why the EP-BT-CR and EP-CR-BT groups are so effective in the original experiment when the program tested in the first session was cmdline and not nametbl, and all the subjects participating in the experiments regarded cmdline as a program that is hard to understand and test. In actual fact, all the above explanations are tentative. As already mentioned, we will not be able to clearly understand the effects of the technique and session/program factors unless further replications are conducted.

5.2 Lessons Learned

The replication that we have conducted is one of a long line of experiments. This means that information and the experience gathered from multiple replications conducted as part of this family is reasonably thorough, and, as we also had access to the original experiment report, the replication could have been carried out without any additional information.

With hindsight, however, we believe that we would have had very little prospect of success if we had proceeded in this manner (as already mentioned, we had intensive communication with the original experimenters). The likenesses between the two experiments are noteworthy, but the differences are no less marked. We have been able to trace these differences back to characteristics of the experimental setting (e.g., software verification and validation course intensiveness), but this was possible only because relatively few changes were made to the replication. For example, we might have ascribed the low effectiveness of the ESPEL subjects using the *branch testing* technique to the fact that they were inexperienced at programming in C. But, UPM subjects are generally not very experienced C programmers either and they are more effective. Therefore, the low effectiveness is more likely to be due to the training received, which does vary, and very much so, from ESPEL to UPM.

With no more than the original experiment report and the experimental materials, such a close coincidence appears to be very hard to achieve. The experimental material did not describe training issues, such as mentioned above; nor did it detail experiment execution or results measurement, for example. Had we not cooperated with the original experimenters, especially during the early stages, the differences between the original experiment and the replication would probably have been much larger. These differences (such as, for example, the above changes regarding training) are also likely to have caused discrepancies in the results. A post-experimental discussion with the original experimenters could show up design differences, but would not be able to prevent any discrepancies caused by such changes. However, these points were discussed at the pre-experimental meetings that we had with the original experimenters.

As a corollary to the above, the experience of having replicated an experiment previously conducted by other experimenters and, especially, the attempt at comparing the results of the two experiments, has shown that unless replications closely resemble the original experiment it is impossible to ascribe (at least hypothetically) consistent and inconsistent results to particular factors and parameters. For the reasons discussed above, if the experimental settings are not alike, the differences between the results can be attributed to virtually any aspect, making the replication much less enlightening than it would otherwise be.

Obviously, merely replicating an experiment with a similar design does not guarantee that the results can be definitely ascribed to a factor. On the one hand, any experiment is subject to some error probability (α and β). On the other hand, we do not know which aspects of a setting (that is, uncontrolled variables) are likely to alter the effects of the factors. Consequently, even very similar replications can return contradictory results.

We can reduce α and β error fairly simply by increasing the number of experimental subjects. As the number of subjects increases, experimenters can gradually lower the α and β values. In practice, however, the availability of experimental subject is limited (for example, Sjøberg et al. 2005, report that the mean number of experimental subjects per SE experiment is 48.6). In other words, we can at best reckon with enough subjects to assure normality and elude small sample effects (from 30 to 50 subjects) (Richy et al. 2004; Graham and Schafer 1999).

The second problem cannot be solved by running a single replication. The unknown variables are, as their name indicates, inscrutable, and therefore their effect cannot be estimated. Nevertheless, randomization is usually considered effective (that is, cancels out the effects of uncontrolled variables) as of 30 subjects, which is when a sample of a standard population is assumed to meet the normality assumption.

The replication that we have conducted has both of the above characteristics. It has 23 experimental subjects, which is lower than the average 48.6 subjects per SE experiment reported by Sjøberg et al. (2005). However, as a result of the cross-over design, those 23 subjects are equivalent to $23 \times 2 = 46$ experimental units. In the case of repeated-measures designs, it is the number of experimental units, not subjects, that influences the α and β values. Sjøberg et al. (2005) do not report the number of experimental units per SE experiment. However, the number is unlikely to be much greater than 48.6, because within-subjects is not the most

common experimental design in SE. Therefore, this replication can be considered an "average" SE experiment.

Even so, we have observed inconsistencies with the original experiment despite having kept all parameters and factors reasonably well under control. We have hypothesized that such inconsistencies can be put down to certain causes (as in the case of the differences in the results with respect to session/program), but this has only been possible because the experiments are quite alike. If there were more differences between the experiments, any such, even hypothetical, attribution would be out of the question.

These may not, of course, be the real causes of the inconsistencies. Literal replications (that is, replications that closely resemble the original experiment) are just a starting point. We will in any case have to run differentiated replications in order to more generally explore all the factors that potentially have an effect.

Finally, we have found that the preparation of the replication accounts for a much larger workload than the experimental sessions. Gaining a detailed understanding of the original experiment, plus the initial training and processing of the forms submitted by subjects, proved to be much more time-consuming than the experimental sessions. In fact, the session workload was comparatively insignificant.

Briefly the lessons learned were:

- Support from the original experimenters is important, during the early replication preparation phases at least, in order to supplement the information available in reports and materials. It is vital to have as much information as possible to ensure that the original experiment and replication are comparable.
- The replication must be as like the original experiment as possible in order to be able to ascribe the detected differences (or similarities) to specific variables.
- The highest experiment workload is spent not on the experimental sessions per se but on the preparation of the experiment and analysis of the information gathered from subjects.

5.3 Experiences with Reporting Guidelines

Apart from reporting the replication, we have, as part of this research, tested the guidelines for reporting replications proposed by Carver (2010). Generally, the guidelines have proved to be really useful, especially as regards the description of the replication in terms of its differences to the original experiment. However, there are some points that we found not to be fully satisfactory and think should be improved:

- The granularity with which the original experiment should be described is unclear. On the one hand, the original experiment can be assumed to have been published, meaning that a brief description would do the job (interested readers could always refer to the original publication). But, on the other hand, the description should be detailed enough for readers to be able to understand the impact of the changes made to the replication. The guidelines should be clearer in this respect.
- Again regarding the reporting of the original experiment, our impression is that the section contents are very unbalanced. The research questions section has very little content, whereas the experimental design section is packed out. Additionally, it is unclear where the hypotheses should be defined. We have moved some

elements (hypotheses, factors and response variables) to the research questions section, but we consider this procedure to be unsatisfactory.
- In the case of literal replications, it is unclear which parts of the experiment or replication to report. In the replication that we have run, for example, we use only two of the three main factor levels of the original experiment. Should we report the results concerning the third level of the original experiment? On the one hand, it appears that we should, otherwise the report would be incomplete. But, on the other hand, the third level is of no use for comparing the original experiment and the replication. Also, readers could always refer to the original experiment report to look up any information about this third factor.
- There is no separate section for discussing the results of the replication. The replication results should be discussed separately before identifying the similarities and differences between experiments. We have added a separate section, but we think that the guidelines should take this point into account.

5.4 Future Work

The replication that we have run has essentially confirmed the effects observed in the original experiment. However, there are some effects, such as differences of effectiveness associated with sessions/programs or carryover effects, which we have still not been able to positively ascribe to specific variables. Our short-term goal is to continue replicating the UPM experiment, altering the setting as little as possible in order to determine beyond all doubt which variables produce which effects. Once we have a good understanding of how the (*equivalence partitioning* and *branch testing*) testing techniques behave, we will be able to run differentiated replications that explore different settings or populations (e.g., experienced professionals or industrial settings).

Acknowledgements This research has been funded by a grant from the Armed Forces Technical School (ESPE), Republic of Ecuador National Higher Education, Science, Technology and Innovation Secretary's Office (SENESCYT) and partially funded by the Spanish Ministry of Economics and Competitiveness project TIN2011-23216.

We also thank the reviewers for their thoughtful review, which greatly improved the quality of the manuscript.

Appendix

Appendix A: Descriptive Statistics

Table 19 Descriptive statistics InScope variable

Technique	N	Mean	Std. dev.
Branch testing	23	31.883	25.581
Equivalence partitioning	23	44.204	29.988

Table 20 Descriptive statistics OutScope variable

Technique	N	Mean	Std. dev.
Branch testing	23	27.536	32.803
Equivalence partitioning	23	11.593	16.231

Appendix B: Survey's Data

Table 21 Survey's data

Subject	Group	Q1	Q2	Q3
S1	G2	3	3	2
S2	G1	NA	NA	NA
S3	G1	5	4	4
S4	G2	5	3	4
S5	G2	4	2	2
S6	G2	5	2	2
S7	G1	4	2	2
S8	G1	5	4	3
S9	G1	5	5	4
S10	G2	5	4	2
S11	G2	4	4	1
S12	G1	4	4	4
S13	G2	NA	NA	NA
S14	G1	4	5	4
S15	G1	3	4	1
S16	G1	4	3	3
S17	G1	4	4	1
S18	G2	4	3	3
S19	G2	5	5	4
S20	G2	4	5	2
S21	G1	5	3	2
S22	G1	5	4	4
S23	G2	4	4	2

Q1: Question 1
Q2: Question 2
Q3: Question 3
Value 1: None
Value 2: I know the theory
Value 3: I have done short exercises
Value 4: Practical assignments
Value 5: Development projects

Appendix C: Replication's Raw Data

Table 22 Replication's raw data

S	G	T	P	Se	Vis. for EP			Vis. for BT			In	Out
					F1	F2	F3	F4	F5	F6	(%)	(%)
1	2	EP	Na	1	1		1				66.7	0.0
1	2	BT	Nt	2					1		33.3	0.0
2	1	BT	Na	1		1					0.0	33.3
2	1	EP	Nt	2	1						33.3	0.0
3	1	BT	Na	1							0.0	0.0
3	1	EP	Nt	2	1						33.3	0.0
4	2	EP	Na	1			1				33.3	0.0
4	2	BT	Nt	2							0.0	0.0
5	2	EP	Na	1	1		1				66.7	0.0
5	2	BT	Nt	2							0.0	0.0
6	2	EP	Na	1			1		1		33.3	33.3
6	2	BT	Nt	2							0.0	0.0
7	1	BT	Na	1							0.0	0.0
7	1	EP	Nt	2	1	1			1		66.7	33.3
8	1	BT	Na	1							0.0	0.0
8	1	EP	Nt	2							0.0	0.0
9	2	EP	Na	1		1	1				66.7	0.0

Table 22 (continued)

S	G	T	P	Se	Vis. for EP			Vis. for BT			In	Out
					F1	F2	F3	F4	F5	F6	(%)	(%)
9	2	BT	Nt	2					1	1	66.7	0.0
10	1	BT	Na	1							0.0	0.0
10	1	EP	Nt	2							0.0	0.0
11	2	EP	Na	1		1	1				66.7	0.0
11	2	BT	Nt	2						1	33.3	0.0
12	1	BT	Na	1	1	1	1				0.0	100.0
12	1	EP	Nt	2	1					1	33.3	33.3
13	2	EP	Na	1							0.0	0.0
13	2	BT	Nt	2		1	1		1		33.3	66.7
14	1	BT	Na	1							0.0	0.0
14	1	EP	Nt	2							0.0	0.0
15	1	BT	Na	1		1					0.0	33.3
15	1	EP	Nt	2						1	0.0	33.3
16	1	BT	Na	1		1		1			33.3	33.3
16	1	EP	Nt	2			1	1			33.3	33.3
17	1	BT	Na	1		1					0.0	33.3
17	1	EP	Nt	2							0.0	0.0
18	2	EP	Na	1							0.0	0.0
18	2	BT	Nt	2				1		1	66.7	0.0
19	2	EP	Na	1	1	1					66.7	0.0
19	2	BT	Nt	2							0.0	0.0
20	1	BT	Na	1							0.0	0.0
20	1	EP	Nt	2					1		0.0	33.3
21	2	EP	Na	1	1	1	1		1		100.0	33.3
21	2	BT	Nt	2	1	1			1		33.3	66.7
22	1	BT	Na	1							0.0	0.0
22	1	EP	Nt	2							0.0	0.0
23	2	EP	Na	1	1	1					66.7	0.0
23	2	BT	Nt	2					1		33.3	0.0

S: Subject, *G*: Group
T: Technique
(*EP*: Equivalence Partitioning, *BT*: Branch Testing)
P: Program (*Na*: Nametbl; *Nt*: Ntree)
Se: Session, *F1–F6*: Faults, *Vis.*: Visible
In: InScope, *Out*: OutScope (Response Variables)

References

Basili VR (1992) Software modeling and measurement: the goal/question/metric paradigm. Tech. Rep. UMIACS TR-92-96, Departament of Computer Science, University of Maryland, College Park

Basili VR, Selby RW (1985) Comparing the effectiveness of software testing strategies. Tech. Rep. TR-1501, Departament of Computer Science, University of Maryland, College Park

Basili VR, Selby RW (1987) Comparing the effectiveness of software testing strategies. IEEE Trans Softw Eng SE-13:78–96

Brown BW (1980) The crossover experiment for clinical trials. Biometrics 36:69–70

Carver JC (2010) Towards reporting guidelines for experimental replications: a proposal. In: Proceedings of the 1st international workshop on Replication in Empirical Software Engineering Research (RESER). Cape Town, South Africa, 4 May 2010

Gómez O (2012) Tipología de Replicaciones para la Síntesis de Experimentos en Ingeniería del Software. PhD thesis, Universidad Politécnica de Madrid

Gómez O, Juristo N, Vegas S (2010) Replications types in experimental disciplines. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement, no. 3. Bolzano-Bozen, Italy, pp 1–10
Graham JW, Schafer JL (1999) On the performance of multiple imputation for multivariate data with small sample size. In: Hoyle RH (ed) Statistical strategies for small sample research. Sage Publications, pp 1–29
Juristo N, Vegas S (2003) Functional testing, structural testing and code reading: what fault do they each detect? In: Empirical Methods and Studies in Software Engineering Experiences from ESERNET, vol 2785(12), pp 208–232
Juristo N, Vegas S, Apa C (2013) Effectiveness for detecting faults within and outside the scope of testing techniques: a controlled experiment. Available at http://www.grise.upm.es/reports.php. Accessed 15 May 2013
Kamsties E, Lott C (1995) An empirical evaluation of three defect-detection techniques. In: Fifth European Software Engineering Conference (ESEC '95). Lecture Notes in Computer Science, vol 989, pp 362–383
Kernan WN, Viscoli CM, Makuch RW, Brass LM, Horwitz RI (1999) Stratified randomization for clinical trials. J Clin Epidemiol 52(1):19–26
Kitchenham B, Fry J, Linkman S (2003) The case against cross-over designs in software engineering. In: Eleventh annual international workshop on software technology and engineering practice, pp 65–67
Meyers LS, Gamst G, Guarino AJ (2006) Applied multivariate reseach: design and interpretation. Sage Publication
Myers GJ (1978) A controlled experiment in program testing and code walkthroughs/inspections. In: Communications of the ACM, vol 21, pp 760–768
Richy F, Ethgen O, Bruyère O, Deceulaer F, Reginster J (2004) From sample size to effect-size: Small study effect investigation (ssei). In: The Internet Journal of Epidemiology, vol 1
Roper M, Wood M, Miller J (1997) An empirical evaluation of defect detection techniques. Inform Softw Technol 39(11):763–775
Senn S (2002) Cross-over trials in clinical research, 2nd edn. Wiley
Sjøberg DI, Han Hannay JE, Hansen O, Kampenes VB, Karahasanovic A, Liborg N-K, Rekdal AC (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31:733–753
Wood M, Roper M, Brooks A, Miller J (1997) Comparing and combining software defect detection techniques: a replicated empirical study. In: Proceedings of the 6th European software engineering conference held jointly with the 5th ACM SIGSOFT international symposium on foundations of software engineering. Zurich, Switzerland, pp 262–277

Cecilia Apa is an Assistant Professor at the Engineering School at the Universidad de la República (UdelaR), coordinator of the Informatics Professional Postgraduate Center (UdelaR) and member of the Organization Committee of the Software and Systems Process Improvement Network in Uruguay (SPIN Uruguay). She received his B.Eng. in Computer Science from the UdelaR. She has several articles published in regional and international conferences. Her main research topics are empirical software engineering and software testing.

Oscar Dieste is research scientist with the Universidad Politécnica de Madrid. Previously, he has been Fulbright Scholar with the University of Colorado at Colorado Springs and assistant professor with the universities Complutense de Madrid and Alfonso X el Sabio. His research interests include empirical software engineering, requirements engineering and their intersections. He received his B.S. in Computing from the University of La Coruña and his Ph.D. from the University of Castilla-La Mancha.

Edison G. Espinosa G. PhD student and Master of software engineering at Universidad Politécnica de Madrid (UPM), Spain. He is full professor of software engineering in Escuela Politécnica del Ejército Sede Latacunga, Ecuador.

Efraín R. Fonseca C. received the MSc degree in 2010. He has ten years of IT industry experience as consultant. He is full professor at Universidad Politécnica del Ejército of Ecuador and now is PhD student at Universidad Politécnica de Madrid. Among his research interests are research process in empirical software engineering, research methods in empirical software engineering, object-oriented analysis and design and ontological representations in software engineering.

Empir Software Eng (2014) 19:558–581
DOI 10.1007/s10664-012-9229-5

An industrial study of applying input space partitioning to test financial calculation engines

Jeff Offutt · Chandra Alluri

Published online: 23 September 2012

Editor: James Miller

Abstract This paper presents results from an industrial study that applied input space partitioning and semi-automated requirements modeling to large-scale industrial software, specifically financial calculation engines. Calculation engines are used in financial service applications such as banking, mortgage, insurance, and trading to compute complex, multi-conditional formulas to make high risk financial decisions. They form the heart of financial applications, and can cause severe economic harm if incorrect. Controllability and observability of these calculation engines are low, so robust and sophisticated test methods are needed to ensure the results are valid. However, the industry norm is to use pure human-based, requirements-driven test design, usually with very little automation. The Federal Home Loan Mortgage Corporation (FHLMC), commonly known as Freddie Mac, concerned that these test design techniques may lead to ineffective and inefficient testing, partnered with a university to use high quality, sophisticated test design on several ongoing projects. The goal was to determine if such test design can be cost-effective on this type of critical software. In this study, input space partitioning, along with automation, were applied with the help of several special-purpose tools to validate the effectiveness of input space partitioning. Results showed that these techniques were far more effective (finding more software faults) and more efficient (requiring fewer tests and less labor), and the managers reported that the testing cycle was reduced from five human days to 0.5. This study convinced upper management to begin infusing this approach into other software development projects.

Keywords Software testing · Industrial study · Input space partitioning

J. Offutt (✉)
Software Engineering, George Mason University, Fairfax, VA, USA
e-mail: offutt@gmu.edu

C. Alluri
Freddie Mac, McLean, VA, USA
e-mail: chandra_alluri@freddiemac.com

1 Introduction

A test criterion is a set of engineering rules that define specific requirements on designing tests, such as cover every branch, or ensuring that every variable definition reaches a use. Although researchers and academics have been publishing test criteria for years, the authors have had difficulty convincing practitioners that the cost of investing in criteria-based test design will lead to better software with acceptable cost. This is a classic return on investment concern: Will the benefits of investing in new technology outweigh the costs? These doubts were expressed by a project manager to a test manager at a large financial services company, the Federal Home Loan Mortgage Corporation (FHLMC), commonly known as Freddie Mac. In response, the test manager proposed to partner with a researcher at a university to choose appropriate test criteria, build support test automation tools, and compare the results of applying test criteria with the results of Freddie Mac's standard test process (manual requirements-based testing). The research question has three simple parts: (1) Can input space partitioning and semi-automated requirements modeling succeed in a real industrial setting with real testers? (2) Can such an approach result in more fault detection during testing, and therefore better software? (3) Can real testers accept this approach for practical use?

Results from the resulting industrial study on four separate software systems are reported here. The project has been very successful. In all four systems, the criteria-based approach yielded **fewer tests** that found **more defects**. All four systems have reported **zero defects** since release. Additionally, the test managers reported that the testing cycle was reduced from **five human days to 0.5**.

This paper reports what we choose to call an "industrial study," rather than a controlled experiment. The study was carried out at an industrial site and we had to play by industrial rules. This is both a strength of the paper and a weakness. This is a strength because this study shows that input space partitioning (ISP) (Ammann and Offutt 2008; Grindal et al. 2005) can be used effectively, with a positive return on investment, in a realistic setting as opposed to a laboratory. But the context also creates a weakness because we were not able to do all the things we would have liked to do. This is common in industrial studies, and we believe the field needs more industrial studies, not fewer.

Financial services like banking, mortgage, and insurance contain subsystems that involve complex calculations. Pricing loans, amortizing loans, asset valuations, accounting rules, interest calculations, pension calculations, and generating insurance quotes are common calculations used by these applications. Calculations embedded into these systems differ in their calculation algorithms. In a particular application, different calculators may need to perform multiple calculations to achieve the business's objective. These calculators together are called the *calculation engine*. In most cases, several calculations need to be performed in sequence or in parallel to get the final output. The logic for these calculations usually resides deep in the business layer of software, which means that system-level inputs must travel through several layers of software and numerous intermediate computations before reaching the financial calculations being tested. This makes it difficult for system testers to control the values of the inputs to the actual financial calculations, that is, *controllability* (Freedman 1991) is low. Likewise, the results of the financial calculations are processed through several layers of software, making it difficult to see the direct

results of the individual financial calculations. That is, *observability* (Freedman 1991) is also low. Software that exhibits low controllability and observability is notoriously hard to effectively evaluate during system testing (Freedman 1991). (These concepts are defined more carefully in the next subsection.)

Financial models are a common form of calculation engine. Financial modeling is the process by which an organization constructs a financial representation of some or all of its financial aspects. The model is built by calculations, and then recommendations are made by using the model. The model may also summarize particular events for the user and provide direction regarding possible actions or alternatives.

Financial models can be constructed by computer software or with a pen and paper. What is most important, however, is not the kind of technology used, but the underlying logic that encompasses the model. A model, for example, can summarize investment management returns, such as the Sortino ratio (Sortino and Price 1994), or it may help estimate market direction, such as the Federal Reserve model (Lander et al. 1997).

It is essential to test financial models thoroughly as they are business critical and may cause enormous harm to the business if wrong. The common system test strategy is to derive test requirements from black box testing techniques such as boundary value analysis, and error guessing. Unfortunately, these are not always effective. Effective test methods need to be used to overcome the calculations' low observability and controllability.

This paper presents an industrial study. Input space partitioning was used to test several major pieces of functionality in large financial calculation engines at a major financial services company (Freddie Mac). As far as we know, this is the first industrial study using input space partitioning. The first author is a test manager in charge of testing these calculation engines and performed this study under the direction of the second author. Section 2 describes some of the key ideas for how calculation engines work. Section 3 describes the testing approaches that were used in this study. Section 4 presents the software systems that were tested and Section 5 gives the testing results. Section 6 provides conclusions and recommendations.

2 Characteristics of Calculation Engines

Calculation logic is implemented in the business layer of multi-layer software systems (usually deployed on local web servers). All calculations are performed on the server; the client is abstracted from the processing. Therefore the user does not observe any processing behind the graphical user interface. For example, a user supplies inputs for an insurance quote and the application generates the insurance quote by performing various calculations on the server. Then the user enters different characteristics of the borrower and the application generates the interest rate by applying different rules on the server. The application takes different inputs from taxpayers and generates the tax owed by performing other calculations on the server. Calculation engines feature some characteristics of component-based applications, reducing their testability.

In general terms, *testability* refers to how hard it is to test a software component (Ammann and Offutt 2008; Freedman 1991; Voas 1992). Testability is largely influenced by two aspects of software, controllability and observability. Ammann

and Offutt (2008) define software observability and controllability as follows. *Software observability* is how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components. *Software controllability* is how easy it is to provide a program with the needed inputs in terms of values, operations, and behaviors. Because calculations are performed on the server, many inputs are taken from other software components as shared through persistent data on disk or in-memory objects, and calculations often depend on the time of the day or day of the month, both observability and controllability are quite low for this software. Problems with observability and controllability are usually addressed by test interfaces or test drivers, which let testers assign specific values to variables during execution, and view values at intermediate steps. Freddie Mac had never used test interfaces before this project.

2.1 Specification Formats for Calculation Engines

Requirements for calculation engines are specified in various forms and in combinations of plain English, use cases, mathematical expressions, logical expressions, business rules, procedural design, and mathematical formulas. These requirements are very complicated for both developers and testers.

Defects in calculation engines not only lead to interruptions, but also can result in legal battles and large financial liabilities. These incidents create headlines in newspapers, causing severe damage to the corporations' reputations. Therefore, strict IT controls are put into place around these applications, and they are subjected to regular auditing.

Although users most commonly see results of financial calculation engines with two digits of decimal precision (dollars and pennies in the USA), most calculations are performed with floating point arithmetic for greater precision. This brings up the possibility of errors in truncation and rounding. Many applications maintain constant word size through the basic arithmetic operations. Multiplication is the biggest concern as multiplying two N-bit data items yields a $2N$-bit product, so truncation limits must be defined in the specifications. Therefore tests must be designed to evaluate precision, truncation, and rounding of the calculated values.

2.2 Characteristics of Design and Implementation of Calculation Engines

Calculation engines have several unusual characteristics that complicate test design, test automation, and test execution. Values such as interest rates, S&P index, NYMEX index, etc. change constantly during a business day depending on market factors. The calculations use some of these values in their computations. These values are updated constantly into tables called *pricing grids*. Calculation systems then pull the current values when needed. When designing tests, this factor can be abstracted or discounted, as this need not be tested every time.

Attributes for calculations are often received from external systems (*upstream*). The systems under test process the calculations and may send the data to external (*downstream*) systems that consume the outcomes. For example, *Asset valuation calculations* receive inputs from *Sourcing* systems and pass the data to the *Subledger* and *General Ledger* downstream systems, where accounting calculations (principles) are applied and the final result will be reflected in financial reports at the end of

the period. A common problem is that the requirements may not clearly specify the source of the data for calculations. Thus, understanding the technical specifications is essential–especially in determining the preconditions and designing *prefix values* (values needed to put the software into the correct state to run the test values).

Understanding the events and conditions that determine the flow in the calculations also helps design effective tests. For example, the Interest Rate type (Fixed, ARM, or Balloon) determines which path to follow. Calculations take different paths based on these inputs.

Algorithms for amortization, pricing, insurance quotations, asset valuations, and accounting principles are standard. For example, amortization methods could be based on the diminishing balance or flat rate over a preset duration. Knowing how these algorithms work is necessary to determine the expected outputs for the tests. For example, MS-Excel has standard amortization functions, which can be used as a calculation *simulator* instead of building simulator programs.

In almost all the applications, most calculations are implemented either as a batch process or an online transaction that occurs in the business layer. Understanding the architecture helps isolate the testable requirements from non-testable requirements.

Even though the entities that participate in the calculations have many important attributes, it is common for only a few to be involved in the calculations. For example, the loan pricing calculation, *Loan* and *Master Commitment*, have 140 and 35 attributes that are available to the calculations, but only seven are actually used in the calculations. Identifying the influential attributes, and their constraints, is necessary to build effective tests. The acceptable values for each attribute and their constraints are defined in the form of business rules. When tests are built, test inputs need to include values for the remaining attributes to make a test case executable.

Calculation engines send and receive values between each other. In many cases, debugging the incorrect output is tedious as it involves checking all intermediate values in the flow. The same set of inputs may yield different outputs when the calculations are performed at different times. The reasons could be: (a) input values are interpreted differently, (b) interest values could be changed in different time periods, (c) intermediate values could have changed, (d) business rules would have changed in the due course, etc. The systems do not store the intermediate values, but intermediate values are essential in diagnosing problems.

Applications that involve these calculations often need to be tested for different business cycles; daily, monthly, quarterly, and annually. Therefore, the same tests may need to be executed more than once.

3 Test Approach

As said in Section 1, calculation engines have low controllability and observability, which makes it more difficult to design and automate complete tests. Depending on the software, the level of testing, and the source of the tests, the tester may need to supply other inputs to the software to affect controllability and observability. Two common practical problems associated with software testing are how to provide the right values to the software, and observing details of the software's behavior. Ammann and Offutt (2008) use these two ideas to refine the definition of a test case as follows. A *prefix value* is any input necessary to put the software into the appropriate state to receive the test case values (related to controllability). A *postfix*

value is any input that is needed after the test case values to terminate the program or see the output (related to observability).

A test case is the combination of all these components (test case values, prefix values, and postfix values), plus expected results. This paper uses "test case" to refer to both the complete test case and test case values.

This study tested the calculation engines using two different methods: input space partitioning and requirements modeling. This was a project decision made by the test manager at the beginning of the project.

3.1 Input Space Partitioning

Input space partitioning (ISP) divides an input space into different *partitions* and each partition consists of different *blocks* (Ammann and Offutt 2008; Grindal et al. 2005). ISP can be viewed as defining ways to divide the input space according to test requirements. The input domain is defined in terms of possible values that the input parameters can have. The input domain is then partitioned into regions that are assumed to contain equally useful values for testing.

Consider a partition q over a domain D. The partition q defines the set of equivalence classes, called blocks B_q. The blocks are pairwise disjoint, that is:

$$b_i \cap b_j = \emptyset,\ i \neq j;\ b_i, b_j \in B_q$$

and together the blocks cover the domain D, that is:

$$\bigcup_{b \in B_q} b = D$$

ISP started with the category partition method (Ostrand and Balcer 1988; Ostrand et al. 1986). Category partition was defined to have six manual steps to identify input space partitions and convert them to test cases.

1. Identify functionalities, called *testable functions*, which can be tested separately.
2. For each testable function, identify the explicit and implicit *variables* that can affect its behavior.
3. For each testable function, identify *characteristics* or categories that, in the judgment of the test engineer, are important factors to consider in testing the function. This is the most creative step in this method whose result will vary depending on the expertise of the test engineer.
4. Choose a *partition*, or set of *blocks*, for each characteristic. Each block represents a set of values on which the test engineer expects the software to behave similarly. Well-designed characteristics often lead to straightforward partitions.
5. Choose a *test criterion* and generate the *test requirements*. Each partition contributes exactly one block to a given test requirement.
6. Refine each test requirement into a *test case* by choosing appropriate values for the explicit and implicit variables.

This project uses several ISP criteria: base choice, multiple base choice, and pairwise.

The *base choice (BC)* criterion emphasizes the most "important" values. A *base choice block* is selected for each partition, and a *base test* is formed by using any value from each base choice for each partition. Subsequent tests are chosen by

holding all but one base choice constant and using each non-base choice in each other parameter. All values in a block are treated identically, so the subsequent discussion sometimes uses the term "block" to refer to the specific value from the block that is used in tests.

For example, if there are three partitions with blocks [A, B], [1, 2, 3], and [x, y], suppose base choice blocks are "A," "1" and "x." Then the base choice test is (A, 1, x), and the following tests would be needed:

(**B**, 1, x)
(A, **2**, x)
(A, **3**, x)
(A, 1, **y**)

A test suite that satisfies BC will have one base test, plus one test for each remaining block for each partition. Base choice blocks can be the simplest, the smallest, the first in some ordering, or the most likely from an end-user point of view. Combining values from more than one invalid block is considered to be less useful because the software often recognizes the value from one block and then negative effects of the others are masked. Which blocks are chosen for the base choices becomes a crucial test design decision. It is important to document the strategy that was used so that further testing can reevaluate that decision.

Sometimes it is difficult to choose just one block as a base choice. The *multiple base choices (MBC)* criterion requires at least one, but allows more than one, base choice block for each partition. Base tests are formed by using each base choice for each partition at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other parameter.

In the *pairwise (PW)* criterion, a value from every block for each partition must be combined with a value from every block for every other partition.

For example, if the model has three partitions with blocks [A, B], [1, 2, 3], and [x, y], then PW will need tests to cover the following combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

Pairwise testing allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, ∼, y)	(B, ∼, x)

The tests with "∼" mean that any block can be used. A test set that satisfies PW testing is guaranteed to pair a value from each block with a value from each other block. In general, pairwise testing does not subsume base choice testing.

3.2 Requirements Modeling

In Freddie Mac's standard testing process, testers develop tests from requirements by informally considering the behavior of the software and guessing what might go wrong. No test criterion is used, no model of the input space or the software is constructed, and there is no notion of coverage. Most tests are not designed before the software is tested; the testers read the requirements, then sit down in front of the software and started running it. Beizer (1990) and Myers (1979) and others extensively discussed this type of behavioral testing from requirements, which allows domain knowledge to be directly used in test design.

As part of this project, we developed a special purpose automated tool called the Fusion Test Modeler (FTM), which helped use the requirements for calculation engines to create a model for generating tests case (a *test model*). FTM also provided traceability from the functional requirements to the test requirements to the tests.

The requirements of the calculation engines are expressed in a mixture of event sequences, action sequences, business rules, use cases, plain text in English, logical expressions, and mathematical expressions. For example, pricing a loan or a contract occurs when some events occur, such as creating the loan, changing the time period, changing the interest rates, and/or changing the fee rates. Amortization calculations depend on the time period of the loan and characteristics of the loan, such as ARM or fixed. Asset valuation triggers a different set of calculations based on the Asset type, e.g., whole loans, swaps, or bonds. Some specifications are defined in the form of pseudo-code and procedural design, especially for financial models, which are often bought as third-party tools and integrated into the Freddie Mac systems. For others, complex calculations are embedded in the sequence of steps in use cases.

The calculation requirements are naturally hierarchical, starting with the overall result needed at the top, then subcalculations, down through individual values at lower levels in the hierarchy. Thus the calculation requirements were modeled for testing as a tree. The test models were extended and decomposed to trace different paths in the models. A typical test requirement is met by visiting a particular node or edge or by touring a particular path. These decomposed paths simplify the complex or obscure behaviors of the calculation engines. Each path in the test models can be refined to a unique test case mapping to the test requirements.

Figure 1 shows the high level process used to test the calculation engines using the modeling technique. The first and second steps were crucial in this process to model the requirements. The Fusion Test Modeler helped model the requirements. The second step derived the test scenarios from the model. FTM automatically generated these test scenarios. Steps 4, 5, and 8 were automated with the help of other tools.

The test modeling process followed 10 steps, as adapted from Beizer (1990).

1. Identify the testable functions (by hand).
2. Examine the requirements and analyze them for operationally satisfactory completeness and self-consistency (by hand).
3. Confirm that the specification correctly reflects the requirements, and correct the specification if it does not (by hand).
4. Rewrite the specification as a sequence of short sentences (using FTM).
5. Model the specifications using FTM.
6. Verify the test model (by hand).
7. Select the test paths (automated by FTM).

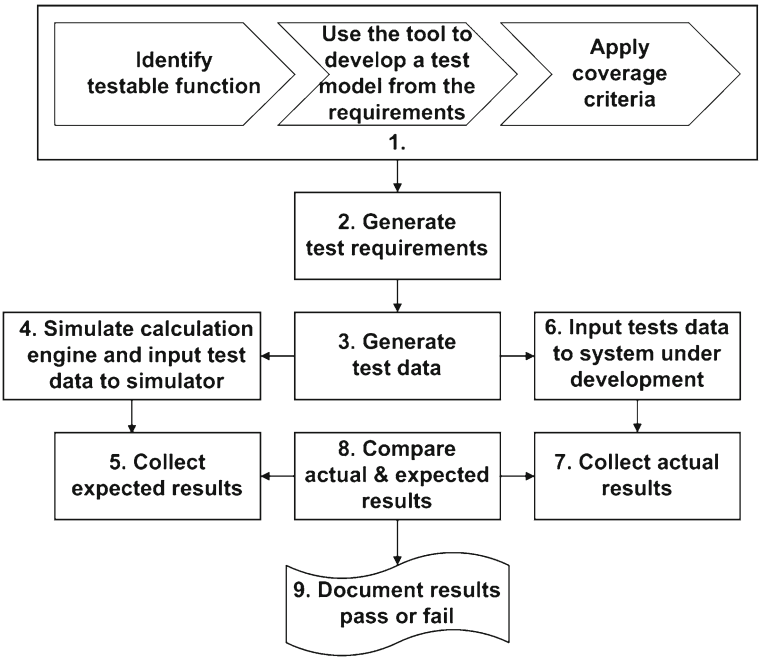

Fig. 1 Modeling process to test calculation engines

8. *Sensitize* the selected test paths; that is, design input values to cause the software to do the equivalent of traversing the selected paths (by hand).
9. Record the expected outcome for each test. Expected results are specified in FTM.
10. Confirm the path (automated by FTM). The prime path coverage criterion (Ammann et al. 2003) is applied to traverse the model's paths.

The algorithms in calculation engines are specified in a variety of formats. Requirements are translated into semi-formal functional specifications. Specifications can be described as finite state machines, state-transition diagrams, control flows, process models, data flows, etc. Financial models are sometimes in the form of the source code, usually when systems are to be built to replicate existing financial models, so the source code becomes the specifications. Sometimes algorithms defined in Visual Basic may be re-implemented in Java, so the Visual Basic version is used as the specification. They are also expressed in logical expressions, use cases, program structures, sequence of events, and sequence of actions.

The tree structure was also used to model logical expressions for testing, as extracted from *if* and *case* statements, and *for* and *while* loops. Multiple-clause predicates were mapped onto a tree structure so that FTM could be used.

UML use cases are also used to express and clarify software requirements. They describe sequences of actions that software performs by expressing the workflow of a computer application. They are often created early and are then used to start test design early. Use cases are usually described textually, but can be expressed as graphs. In this project we expressed use cases as graphs, then selected paths to

embed in trees for use by FTM. These graphs can be viewed as *transaction flows* (Beizer 1990). Activity diagrams can also be used to express transaction flows. FTM can be used to model a variety of things, including state behavior, returning values, and computations.

3.3 The Fusion Test Modeler

FTM was developed to meet seven essential needs.

1. It provides traceability from the requirements to the test models to the tests.
2. It helps testers satisfy internal audit requirements. The testing process must be transparent, the test cases must be well documented, and changes should be applied in a controlled manner. FTM allows test analysts to keep track of changes, and also captures who executed the tests and when they were executed. Models are saved in XML files that are under configuration management.
3. It allows multiple test specification formats.
4. It must be easy to learn with a minimum of training. The modeling technique chosen is simple so that the business community, testers, and analysts from non-engineering backgrounds can learn and model the requirements quickly. They can also analyze the requirements with the help of models.
5. FTM must preserve the mental models used to create the test requirements. Testers often build mental models and then destroy them once they understand the requirements. FTM allows users to build rough drafts of the test models and preserve them for future analysis. The tool helps the users evolve their analysis into a model that captures the testable requirements. It also supports impact analysis when changes need to be made to the software, and helps transition knowledge when new team members arrive.
6. It must complement existing tools used to manage testing.
7. FTM must satisfy graph-based coverage criteria (in this case, all paths in the tree).

FTM stores test requirements in a spreadsheet, and uses Java utilities to read and generate the base choice and multiple base choice test requirements from the spreadsheet. The pairwise test requirements were generated by a PERL program (Bach 2005). Values were obtained from upstream software components and by hand. A simulator, written as Excel functions, was used to generate the expected results. A disadvantage of simulators is that it is difficult to judge whether the output of the simulator or the output of the system-under-test is correct. Differences must be resolved by a domain expert. A second disadvantage is of the same error appearing in both the simulator and the system-under-test.

Rational TestManager stores test data in *data pools*. A data-driven testing technique was applied to automatically enter the test data into the system by the tool. Logic validation was not added to the automation scripts to maximize the processing time of the data entry. Automation scripts were just simulated to enter the data and were scheduled on different machines to enter data in parallel. When the test data was input to the system, calculation-triggering events were identified and automation scripts trigger the calculations. Events to trigger the calculations were also incorporated into the script, so that every time the event triggers, the calculation

engine was activated and performs calculations at the business layer, storing the results in the database.

All actual results were stored in a database. In general, the final state of the actual results generated by the calculation engines were stored in the database, and internal states may be logged into execution logs for later debugging. It may be required to refer to the execution logs for the internal states and values of the actual results if they deviate from the expected results. One of our application study used nine calculators and each calculator received the inputs from one or more of the other calculators. We suggested to the programmers that they generate the execution logs with the intermediate values of the calculation variables to help debug incorrect expected output. A Java utility was written to search all the intermediate states of calculation variables. The program scanned 10 MB of the execution logs in about 10 seconds and wrote the expected intermediate outputs into an Excel spreadsheet.

Financial calculations often produce hundreds of outputs that need to be compared frequently, thus an automated comparison tool was developed to examine and compare the backend results with the spreadsheet. The comparator compares the results, showing the differences for failures and successes for passes. The comparator compares the left-hand side and right-hand side of the results in different forms: spreadsheet to spreadsheet, spreadsheet to database, and spreadsheet to text file.

Sometimes the actual results (intermediate) are obtained from the program execution logs. These logs store values for intermediate results and final results are stored in the database. The comparator searches for the desired text in the execution logs and required fields in the database. The comparator tool discards unneeded text strings before making comparisons of the output results. Actual and expected results may not always be exactly the same due to roundoff, so the expected outputs include *tolerance* limits. For example, a variation of at most one dollar in a million is acceptable if the variation is caused due to drifts in floating point accuracy.

4 Software Systems Studied

This paper presents results from testing four separate industrial systems. They are described here, and results for each are given in the next section. All are complicated financial calculation engines that perform operations that may not be familiar to the readers. More details are in Alluri's MS thesis (Alluri 2008). The test criteria were not applied in a comparative manner, but in a complementary manner, so for example, pairwise testing was used for particularly complicated subsystems and to handle conflicts between partitions. The specific test criteria used depended on characteristics of the systems. This paper shows details of the test designs for the first software system, but omits those details for the other systems to save space. We have not been able to find other industrial studies using input space partitioning.

4.1 Contract Pricing

Contract pricing prices contracts when contracts are created in the Loan Purchase Contract (LPC) subsystem and reprices the contracts when contracts are modified or upon user requests. Two types of contracts are cash contracts and swap contracts. This system tested swap contracts. The requirements for the pricing calculations of

swap contracts are specified in the form of use cases. This use case calculates the swap *GFee*, *Buyup* max, *Buydown* max, *Total adjusted GFee* for fixed rate, *Guarantor*, and *Multilender ARM* swap contracts.

This project tested the software in two stages. The first stage tested the larger *import contracts* feature. The second stage tested a smaller number of contract attributes that were isolated to test just the *contract pricing* feature. Freddie Mac's selling system consists of different subsystems: *LPC*, *NCM*, *TPA*, *Pooling*, *Pricing*, and *OIM*. Each subsystem contains multiple features and is designed to abstract their functionalities from the others. The *contract pricing* feature (stage 2) receives inputs from the *import contracts* feature (stage 1) of the LPC subsystem that facilitates importing the contracts. The *import contracts* feature had almost 200 business rules, and stage 1 testing resulted in 92 base choice and 207 pairwise tests.[1] The stage 2 testing resulted in 15 base choice, 30 multiple base choice, 23 pairwise tests, and 27 requirements modeling tests. For space reasons, this paper gives more test details for the stage 2 testing than stage 1.

In the first stage (important contracts), 29 attributes were identified and used to create 29 partitions for input space partitioning. The blocks for each partition were based on the system specifications and are shown in Table 1. Tests were designed using the base choice coverage criterion and constraints among the partitions were validated using the pairwise coverage criterion.

In the second stage (*contract pricing*), partitions required for just the contract pricing calculations were separated and then the base choice, multiple base choice, and pairwise criteria were applied. Problem analysis showed that of the inputs defined earlier, only seven inputs, *Rate option*, *GFee*, *Remittance option type*, *GFee grid remittance*, *LLGFee eligibility*, *BUBD eligibility*, and *Max Buyup*, control the calculations. Therefore, the other partitions were not considered. The partitions and blocks for *contract pricing* are shown in Table 2. Base choices are highlighted in **bold**.

Base Choice Tests The base choice tests are shown in Table 3. There is one base choice test (test #1), and then one test for each non-base block (14). In the non-base choice tests, the non-base choice values are italicized.

Multiple Base Choice Tests Multiple base choice (MBC) was also used in the second stage for *contract pricing*. Table 4 shows these tests. The first base choice test is the same as with BC, but a second base choice test was added (test #16). With MBC and two base choice tests, exactly twice as many tests are needed.

Pairwise Tests Pairwise testing was used to test constraints among the parameters. This resulted in 23 tests, as shown in Table 5. The "∼" means that the indicated value **cannot** be used.

Requirements Modeling The testable function for *contract pricing* was modeled using the FTM tool. The *contract pricing* calculation simulator was built in Java. This

[1]We used Bach's PERL program to generate pairwise test requirements (Bach 2005). This is probably more tests than necessary and more modern tools, such as NIST's ACTS (Kacker and Kuhn 2008), would probably create far fewer tests.

Table 1 Contract partitions and blocks

Partition	Partition name	Blocks
1	Execution option	GU, ML, NULL_EO, *EO
2	Rate option	FI, AR, NULL_RO, *RO
3	Master commitment	9CHAR, 10CHAR, 8CHAR, NULL_MC, TBD
4	Security product	NUMBER, NULL_SP, *SP
5	Security amount	DOLLAR_ROUND, *DOLLAR_FRACTION, *>100B, NULL_SA
6	Contract name	CHAR (26), CHAR (25), CHAR (1), NULL_CONT
7	Settlement date	MMDDYYYY, *SD, NULL_SD
8	Settlement cycle days	1, 3, 4, 5, *6, *2, NULL_SCD
9	Security coupon	XX.XXX, XXX.XX, NULL_SC, 26.000
10	Servicing option	RE, CT, *SO, NULL_SO
11	Designated servicer number	NULL_DS, DS, *DS
12	Minimum required servicing spread	XX.XXX, NULL_MRSS, XXX.XX
13	Minimum servicing spread coupon	XX.XXX, NULL_MSSC, XXX.XX
14	Minimum servicing spread margin	XX.XXX, NULL_MSSM, XXX.XX
15	Minimum servicing spread lifetime ceiling	XX.XXX, NULL_MSSLC, XXX.XX
16	Remittance option	AR, SU, FT, GO, *RT, NULL_RT
17	Super ARC remittance due day	0, 1, 2, 14, 15, 16, 30, NULL_SARD
18	Required Spread GFee	NULL_RSG, *RSG, RSG
19	BUBD program type	CL, NL, LL, *BUBD_PT, NULL
20	BUBD request type	NULL_BUBD_RT, BO, BU, BD, NO, *BUBD_RT
21	Contract level Buyup/Buydown	NULL_CL_BUBD, *CL_BUBD, BU, BD, NO
22	BUBD grid type	NULL_BUBD_GT, *BUBD_GT, A, A-minus, negotiated 1 grid
23	BU max amount	0, 1, *BU_MAX_AMT, NULL_BU_MAX_AMT, XXX.XXX
24	BD max amount	0, 1, *BD_MAX_AMT, NULL_BD_MAX_AMT, XXX.XXX
25	Pool number	NULL_PNO, PNO, *PNO
26	Index look back period	NULL_ILP, *ILP, ILP
27	Fee type	FT, *FT, NULL_FT
28	Fee payment method	Delivery fee, GFee add on, *FTM, NULL_FTM
29	Prepayment penalty indicator	Y, N

simulator program reads inputs from the spreadsheet, performs the calculations, and then outputs the results into another spreadsheet. This resulted in 27 tests, as shown in Table 6.

Running the Tests All tests, both ISP and requirements modeling tests, were given to the calculation simulator. The calculation simulator performs the calculations and generates expected results for each test input, then writes them into a spreadsheet.

All tests were input to the system-under-test using Rational's robot tool (IBM 2011). The system has a feature called *import contracts* that allows all tests to be

Table 2 Contract pricing partitions and blocks

Partition	Partition name	Blocks
1	Rate option	**Fixed**, ARM
2	GFee	**NotNull**, null
3	Remittance option type	**Gold**, FirstTuesday, ARC, SuperARC
4	GFEE grid remittance option	**Gold**, FirstTuesday, ARC, SuperARC
5	MC LLGFee eligibility	**Y**, N
6	BUBD eligibility	**Prohibited**, required, optional
7	Max Buyup	<**12.5**, =12.5, >12.5, NULL

bundled into a flat file and imported at once. When the contract is created, the system automatically prices the contracts and stores the pricing results in the database as the actual results.

4.2 Loan Pricing

The Loan Pricing feature prices loans when they are newly created or after business users request a reprice. Price recalculations for swap loans are triggered by data corrections to one or more data elements used in the price calculation. These data corrections can be one or both of the internal FM price definition terms (grid data), or seller delivered loan/contract data for fields that affect the price. Either type of data correction will trigger a total price recalculation of all price components that apply to the loan, including GFEE/LLGFEE, BUBD and Delivery Fees. The price recalculation can be approved either automatically or by hand. Any data change

Table 3 Contract pricing stage 2 base choice tests

Test #	Rate option	GFee	Remittance option type	GFEE grid remittance option	MC LLGFee eligibility	BUBD eligibility	Max Buyup
1 *Base*	ARM	NotNull	Gold	Gold	Y	Prohibited	<12.5
2	Fixed	*Null*	Gold	Gold	Y	Prohibited	<12.5
3	Fixed	NotNull	*FirstTuesday*	Gold	Y	Prohibited	<12.5
4	Fixed	NotNull	*ARC*	Gold	Y	Prohibited	<12.5
5	Fixed	NotNull	*SuperArc*	Gold	Y	Prohibited	<12.5
6	Fixed	NotNull	Gold	*FirstTuesday*	Y	Prohibited	<12.5
7	Fixed	NotNull	Gold	*ARC*	Y	Prohibited	<12.5
8	Fixed	NotNull	Gold	*SuperArc*	Y	Prohibited	<12.5
9	Fixed	NotNull	Gold	Gold	*N*	Prohibited	<12.5
10	Fixed	NotNull	Gold	Gold	Y	*Required*	<12.5
11	Fixed	NotNull	Gold	Gold	Y	*Optional*	<12.5
12	Fixed	NotNull	Gold	Gold	Y	Prohibited	*=12.5*
13	Fixed	NotNull	Gold	Gold	Y	Prohibited	*>12.5*
14	Fixed	NotNull	Gold	Gold	Y	Prohibited	*NULL*
15	Fixed	NotNull	Gold	Gold	Y	Prohibited	*<12.5*

Table 4 Contract pricing stage 2 multiple base choice tests

Test #	Rate option	GFee	Remittance option type	GFEE grid remittance option	MC LLGFee eligibility	BUBD eligibility	Max Buyup
1 *Base*	Fixed	NotNull	Gold	Gold	Y	Prohibited	<12.5
2	*ARM*	NotNull	Gold	Gold	Y	Prohibited	<12.5
3	Fixed	*Null*	Gold	Gold	Y	Prohibited	<12.5
4	Fixed	NotNull	*FirstTuesday*	Gold	Y	Prohibited	<12.5
5	Fixed	NotNull	*ARC*	Gold	Y	Prohibited	<12.5
6	Fixed	NotNull	*SuperArc*	Gold	Y	Prohibited	<12.5
7	Fixed	NotNull	Gold	*FirstTuesday*	Y	Prohibited	<12.5
8	Fixed	NotNull	Gold	*ARC*	Y	Prohibited	<12.5
9	Fixed	NotNull	Gold	*SuperArc*	Y	Prohibited	<12.5
10	Fixed	NotNull	Gold	Gold	*N*	Prohibited	<12.5
11	Fixed	NotNull	Gold	Gold	Y	*Required*	<12.5
12	Fixed	NotNull	Gold	Gold	Y	*Optional*	<12.5
13	Fixed	NotNull	Gold	Gold	Y	Prohibited	*=12.5*
14	Fixed	NotNull	Gold	Gold	Y	Prohibited	*>12.5*
15	Fixed	NotNull	Gold	Gold	Y	Prohibited	*Null*
16 *Base*	ARM	NotNull	SuperArc	Gold	N	Prohibited	=12.5
17	*Fixed*	NotNull	SuperArc	Gold	N	Prohibited	=12.5
18	ARM	*Null*	SuperArc	Gold	N	Prohibited	=12.5
19	ARM	NotNull	*Gold*	Gold	N	Prohibited	=12.5
20	ARM	NotNull	*FirstTuesday*	Gold	N	Prohibited	=12.5
21	ARM	NotNull	*ARC*	Gold	N	Prohibited	=12.5
22	ARM	NotNull	SuperArc	*FirstTuesday*	N	Prohibited	=12.5
23	ARM	NotNull	SuperArc	*ARC*	N	Prohibited	=12.5
24	ARM	NotNull	SuperArc	*SuperArc*	N	Prohibited	=12.5
25	ARM	NotNull	SuperArc	Gold	*Y*	Prohibited	=12.5
26	ARM	NotNull	SuperArc	Gold	N	*Required*	=12.5
27	ARM	NotNull	SuperArc	Gold	N	*Optional*	=12.5
28	ARM	NotNull	SuperArc	Gold	N	Prohibited	*<12.5*
29	ARM	NotNull	SuperArc	Gold	N	Prohibited	*>12.5*
30	ARM	NotNull	SuperArc	Gold	N	Prohibited	*Null*

to loan and/or delivery fee data will trigger a recalculation and reprice all price component data that are effective at the time of settlement. This includes any changes to BUBD or contract GFEE grid definition terms.

The mortgage loan entity has nearly 150 attributes, but only a few are relevant to Loan Pricing. Twelve partitions were identified in this testable function. Two of the 12 are received from the price grids. These values are updated in the grids based on the current market. Three others are intermediate parameters whose values are used in the final calculations. Even though they participate in the calculations, their values depend on the values of the other attributes that are inputs. (This is an example of the controllability problem in these applications.)

Among the 12 partitions, only six influence the controllability of the pricing calculations. The remaining six influence observability. Test cases were derived for the base choice (26 tests), the multiple-base choice (52 tests), and the pairwise

Table 5 Contract pricing stage 2 pairwise tests

Test #	Rate option	GFee	Remittance option type	GFEE grid remittance option	MC LLGFee eligibility	BUBD eligibility	Max Buyup
1	Fixed	NotNull	Gold	Gold	Y	Prohibited	<12.5
2	ARM	Null	FirstTuesday	Gold	N	Required	=12.5
3	Fixed	Null	FirstTuesday	FirstTuesday	Y	Optional	<12.5
4	ARM	NotNull	Gold	FirstTuesday	N	Prohibited	=12.5
5	Fixed	NotNull	ARC	ARC	N	Required	>12.5
6	ARM	NotNull	SuperArc	ARC	Y	Optional	Null
7	Fixed	Null	SuperArc	SuperArc	N	Prohibited	>12.5
8	ARM	Null	ARC	SuperArc	Y	Required	Null
9	ARM	Null	Gold	ARC	N	Required	<12.5
10	Fixed	NotNull	FirstTuesday	SuperArc	Y	Optional	=12.5
11	ARM	~Null	Gold	Gold	Y	Optional	>12.5
12	Fixed	~NotNull	FirstTuesday	FirstTuesday	N	Prohibited	Null
13	~ARM	~NotNull	ARC	FirstTuesday	N	Optional	>12.5
14	~Fixed	~Null	ARC	ARC	~Y	Prohibited	=12.5
15	~Fixed	~NotNull	SuperArc	Gold	~N	Required	Null
16	~ARM	~NotNull	SuperArc	SuperArc	~N	~Prohibited	<12.5
17	~Fixed	~Null	SuperArc	FirstTuesday	~Y	Required	>12.5
18	~Fixed	~Null	Gold	Gold	~N	~Optional	Null
19	~ARM	~NotNull	ARC	Gold	~Y	~Prohibited	<12.5
20	~ARM	~NotNull	FirstTuesday	ARC	~Y	~Required	=12.5
21	~Fixed	~Null	Gold	SuperArc	~N	~Optional	=12.5
22	~ARM	~Null	FirstTuesday	~FirstTuesday	~Y	~Prohibited	>12.5
23	~ARM	~Null	SuperArc	~ARC	~N	~Optional	=12.5

coverage criteria (72 tests). The requirements model approach was used to generate 131 tests, many of which were redundant because the same flow of information is duplicated for Fixed, ARM and Balloon contracts. More details about the Loan Pricing test designs can be found in Alluri's MS thesis (Alluri 2008).

4.3 Amortization

The amortization calculator is a modular software component that calculates the amortized cash flows for a given loan. Calculating the loan amortization requires 11 steps.

This system is an example of how different calculations will be triggered based on preceding conditions. A total of 15 calculations follow one another in a sequence and feed their outputs to the following calculator. Five are preliminary calculations. The remaining 10 execute recursively until the end of the loan's term. For example, the ending balance of the loan changes from month to month, e.g., if the loan's life is 30 years, the loan will have 360 installments and when amortized it will have 360 records with varying ending balances for each month. For a given loan, the same types of calculations occur 360 times. Therefore, when defining the scope of each testable function, the loop is considered as one partition and critical characteristics of loops are included as the blocks.

The system has 160 attributes, but only 14 contribute to the calculations. All 15 calculations were treated as testable functions. The total number of base choice

Table 6 Contract pricing stage 2 requirements modeling tests

Test #	Rate option	GFee	Remittance option type	GFEE grid remittance option	MC LLGFee eligibility	BUBD eligibility	Max Buyup
1	Fixed	NotNull	Gold	Gold	Y	Prohibited	>12.5
2	Fixed	NotNull	Gold	Gold	Y	Prohibited	≤12.5
3	Fixed	NotNull	Gold	Gold	Y	Prohibited	>12.5
4	Fixed	NotNull	Gold	Gold	Y	Prohibited	≤12.5
5	Fixed	NotNull	Gold	SuperArc	Y	Prohibited	>12.5
6	Fixed	NotNull	Gold	SuperArc	Y	Prohibited	>12.5
7	Fixed	NotNull	Gold	FirstTuesday	Y	Prohibited	≤12.5
8	Fixed	NotNull	Gold	ARC	Y	Prohibited	≤12.5
9	Fixed	NotNull	Gold	FirstTuesday	Y	Prohibited	≤12.5
10	Fixed	NotNull	Gold	FirstTuesday	Y	Prohibited	>12.5
11	Fixed	NotNull	Gold	ARC	Y	Prohibited	≤12.5
12	Fixed	NotNull	Gold	FirstTuesday	Y	Prohibited	≤12.5
13	Fixed	NotNull	Gold	Gold	N	Prohibited	>12.5
14	Fixed	NotNull	Gold	Gold	N	Prohibited	≤12.5
15	Fixed	NotNull	Gold	SuperArc	N	Prohibited	>12.5
16	Fixed	NotNull	Gold	SuperArc	N	Prohibited	≤12.5
17	Fixed	NotNull	Gold	SuperArc	N	Prohibited	>12.5
18	Fixed	NotNull	Gold	SuperArc	N	Prohibited	≤12.5
19	ARM	NotNull	FirstTuesday	FirstTuesday	Y	Prohibited	>25
20	ARM	NotNull	FirstTuesday	FirstTuesday	Y	Prohibited	≤25
21	ARM	NotNull	FirstTuesday	ARC	Y	Prohibited	>25
22	ARM	NotNull	FirstTuesday	ARC	Y	Prohibited	≤25
23	ARM	NotNull	FirstTuesday	SuperArc	Y	Prohibited	>25
24	ARM	NotNull	FirstTuesday	SuperArc	Y	Prohibited	≤25
25	ARM	NotNull	FirstTuesday	FirstTuesday	N	Prohibited	>12.5
26	ARM	NotNull	FirstTuesday	ARC	N	Prohibited	=12.5
27	ARM	NotNull	FirstTuesday	ARC	N	Prohibited	=12.5

tests is 74. The multiple base choice coverage criterion did not offer any additional coverage, as the partitions are the same for all the instruments. Thus MBC was not used for this system. The blocks had no constraints among them, so the pairwise coverage criterion also did not offer any additional coverage, and was not used. In addition, the FTM tool was not available when this system was tested, so the modeling technique was not used in the Amortization system. More details about the Loan Pricing test designs can be found in Alluri's MS thesis (Alluri 2008).

4.4 Static Effective Yield

Specifications to calculate the Static Effective Yield (SEY) are described in the form of use cases. This calculation is used in GO Amortization to calculate SEY amortization for pools and in segments reporting to calculate SEY amortization for cohorts of whole loans. Amortization calculation functions are recursive in nature.

The use case document had nine sections, but only the two with functional requirements were used in this system. The testing team identified eight testable functions.

Applying the base choice coverage criterion yielded 64 tests. The multiple base choice coverage criterion did not offer any additional coverage, so was not used for

this system. The blocks had no constraints among them, so the pairwise coverage criterion also did not offer any additional coverage, and was not used.

The requirements were classified into eight testable functions. For the modeling technique, the requirements were grouped into three testable functions, producing 12 test cases. More details about the Loan Pricing test designs can be found in Alluri's MS thesis (Alluri 2008).

5 Results

The studies documented here only represent part of the complete set of software systems on which this approach was applied, but the results were similar on other software components. For example, the Contract Pricing and Loan Pricing systems belong to the Selling System, which has about 1200 Java files.

This study measured two things; the ability of the tests to find faults, and coverage of the tests. Results on these are described in the following subsections.

5.1 Fault Detection

All faults were naturally occurring and we did not know *a priori* how many total faults were in the software. The programs' correctness were determined by comparing the outputs of the system-under-test and a simulator. Fault detection was not recorded for the stage 1 tests, so only results from stage 2 tests are given. Faults found for all tests on the four systems are shown in Table 7.

From these data, it is clear that the criteria-based tests found far more faults than the requirements-based tests. Just considering the two systems that used requirements-based tests, the criteria-based tests found 14, 17, and 23 faults, whereas the RM tests only found 7. The specific faults found were all cumulative, that is, all the faults found by RM were also found by BC, all the faults found by BC were also found by MBC, and all the faults found by MBC were also found by PW. After seeing these results, the program manager refused funding for further RM tests. This was a business decision that we had to respect, even though we would prefer to have more data.

Although we were not able to capture the human costs of creating these tests (which are affected by so many factors that the results would hardly be generalizable anyway), the managers reported that the testing cycle was reduced from five human days to 0.5.

We can also take the number of tests as a rough measure of cost. A simple way to estimate *test efficiency* of set of tests is to divide the number of faults found by

Table 7 Faults found by all test sets, including stage 1 and stage 2

Software system	BC tests	Faults found	MBC tests	Faults found	PW tests	Faults found	RM tests	Faults found
Contract pricing	15	6	30	7	230	12	27	3
Loan pricing	26	8	52	10	72	11	131	4
Amortization	74	18	N/A		N/A		N/A	
Static effective yield	64	17	N/A		N/A		N/A	
Total	179	49	82	17	302	23	158	7

Table 8 Fault efficiency – all four studies

Criterion	Tests	Faults	Efficiency
BC	179	49	0.27
MBC	82	17	0.21
PW	302	23	0.08
RM	158	7	0.04

the number of tests. Table 8 shows that all four criteria-based design techniques were far more efficient than the requirements modeling approach. Recall that we cannot compare the total numbers for BC with the other criteria because it was applied to all four studies. These data are also not generalizable because of the small sample sizes. Nevertheless, these data convinced management at Freddie Mac of the positive return on investment for criteria-based testing and automation. We know of no industry standard for the percentage of tests that are expected to find faults, but the test managers at Freddie Mac were shocked at these numbers. Based on their experience, they expected about 5 % of the tests to reveal a fault, and considered 10 % efficiency to be outstanding (or a sign of very poor software).

Further analysis has revealed that the tool used to create pairwise tests was somewhat inefficient. In fact, NIST's ACTS pairwise tool (Kacker and Kuhn 2008) created only 17 tests in stage 1 for the *Contract Pricing* system. This would change the total number of tests from 230 to 40, and if those tests found the same number of faults, the efficiency would be over 50 %. Of course, we are not able to run those tests on the same software, so we cannot know whether a similar number of faults would be found.

We also believe that the data from the MBC and PW tests emphasize that the extra work will find more faults, but with higher cost. Thus the strategy we used of bringing in the stronger criteria when the extra expense is deemed necessary, was validated.

Perhaps the strongest result, however, came after the software was completed and deployed. During the final system testing of these projects, 17,000 records were run and zero defects were detected. This had never happened with any Freddie Mac software before, and this was the first system to go into production with zero non-conformances. In the years since this project finished (in 2008), ZERO faults have been detected in the software tested.

This might be a little surprising in the systems where MBC and PW were not used, since they found additional faults when they were used. But testing stopped with BC when analysis of the input domain model (the partitions and blocks) indicated MBC and PW would not improve testing. So we would not expect many additional faults to be found by stronger criteria in those systems. On the other hand, these systems could have faults that simply have not been revealed as failures yet.

5.2 Coverage Measurement

Two types of coverage measures were used to determine the effectiveness of testing: functional coverage and structural coverage. In this paper, *functional coverage* is a measure of the number of **functional requirements** executed, and *structural coverage* is a measure of the **code statements** executed (LOC). We used the requirements traceability matrix (RTM), which is the list of requirements and the tests that tested

Table 9 Statement coverage results

Software system	LOC	BC	Cover (%)	MBC	Cover (%)	PW	Cover (%)	RM	Cover (%)
Contract pricing									
SwapContractService	258	15	86	30	92	23	92	27	92
SwapContractCalculator	166	15	85	30	90	23	90	27	82
Loan pricing	882	26	86	52	89	72	92	131	97
Amortization	3254	74	100						
Static effective yield	1574	56	100						

each, to evaluate functional coverage and Parasoft's jTest[2] to evaluate structural coverage. jTest offers statistics for statement and method coverage (but not branch, for example). Testers did not have access to the source code, so we relied on developers to help us gather the structural coverage.

Table 9 shows the statement coverage for the stage 2 tests on all four systems, broken into four separate sections for each system. The coverage on the two major components of Contract Pricing are shown separately, although the same tests were used on both.

Table 10 shows the functional requirements coverage for the stage 2 tests on all four systems studied, broken into four separate sections for each system. All tests achieved 100 % functional requirements coverage.

Contract Pricing had 89 requirements for business rules, 22 system-specific requirements, and 92 requirements to generate error messages, for a total of 203 requirements. It had an additional 22 requirements for different combinations of the attributes. The BC tests covered all 203 requirements and 8 of 22 combination requirements. The other combination requirements were covered by the pairwise tests.

The Loan Pricing requirements were captured in use cases that have one main flow, one alternate flow, and three exception flows. The BC, MBC, and PW tests all covered 100 % of the functional requirements.

5.3 Observations

After testing was completed, we asked the testers and managers informally about their opinions of the process and the results. The testers all agreed that the PW criterion is less useful when the characteristics have a large number of attributes because it is difficult to map the PW tests to the requirements when traceability is important. However, the pairwise criterion definitely helps reduce or eliminate the duplicate pairs of inputs and hence is used to eliminate the constraints that do not coexist. If the implementation is such that it will not allow these combinations to be input, then almost all of the pairwise tests become infeasible. Grindal et al. (2007) proposed a *submodel strategy* to handle constraints, which was later found to be more useful for this problem than using PW directly as in this system. Newer tools such

[2] http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

Table 10 Functional requirements coverage results

Software system	BC	Cover (%)	MBC	Cover (%)	PW	Cover (%)	RM	Cover (%)
Contract pricing	15	100	30	100	23	100	27	100
Loan pricing	26	100	52	100	72	100	131	100
Amortization	74	100						
Static effective yield	56	100						

as NIST's ACTS (Kacker and Kuhn 2008) can include constraints during test data generation, making PW even simpler to apply. Although the pairwise criteria was able to cover the 16 requirements that MBC could not, it took a very long time to filter the tests from all the PW tests.

The attributes for Loan Pricing had many constraints. The PW tests gave good coverage, but with a lot of tests. As noted previously, this may be an artifact of the tool used to compute pairwise. PW often has fewer tests than BC. Generally, the number of tests needed for BC is proportional to the number of partitions, whereas the number of tests needed for PW is only *log* the number of partitions (Ammann and Offutt 2008; Grindal et al. 2005). To manually determine which PW tests filled the gaps left by BC took very long time. Most of the requirements modeling tests were redundant because the same information flow is duplicated for Fixed, ARM, and Balloon loans. The requirements model generated 131 tests, many of which were redundant because the same information flow was duplicated for three different kinds of contracts.

Initially, 12 requirements tests were designed for the Static Effective Yield study, but they were flawed in a way that would have made them very expensive to automate.

5.4 Threats to Validity

A study like this has several threats to validity. Most obviously, the study was within one company on a particular kind of software. Thus we cannot be sure that the success would be duplicated in other settings. Another potential validity threat is the FTM tool used in the study, which could have been flawed. Great care was taken to test FTM and the models and resulting tests were spot-checked for accuracy. If FTM was flawed, it seems likely the resulting tests would be less effective, thus this would be a bias against the results presented in this paper. Also, at certain points in the process (as described in Section 3) human testers had to make decisions. It is possible that different testers would have different results. Taken together, these threats mean that we cannot conclude that this type of testing will succeed in all settings. Rather, we know that it is possible for this type of testing to improve testing and lead to higher quality software in some settings.

6 Conclusions and Future Work

This paper shows how high-end, criteria-based, semi-automated test design and implementation can have a strong positive impact on testing in industry. The company,

Freddie Mac, depends on software for success in all aspects of its business and the quality of its software is a primary factor in the success of the company. Problems with the software can result in loss of very large amounts of money. After testing was completed, we asked the testers and managers informally about their opinions of the process and the results. All parties involved, including test management, testers, developers, development managers, and upper management, agreed that this testing process helped create tests that were more effective and with less cost. As a result of this industrial study, these ideas are being infused into software development and software testing is being improved throughout the company. As far as we know, nobody has reported on the use of input space partitioning in an industrial setting before.

As additional analysis, we analyzed post-testing defects in the previous eight releases for the software used in systems 1 and 2. The analysis showed that the testing approaches used in this study would have eliminated 75 % of the post-delivery defects.

The overriding advantage of using ISP (a criterion-based) approach was not surprising: we were able to generate fewer tests that were more effective, and do it more efficiently. The ISP method does not require a strong background in math or computer science, both of which are often short in software testing teams. The ISP method also has a very clear, structured, process to follow, which the testers reported being very comfortable with. We were pleased to find that the ISP tests gave good coverage of both requirements and source code. It was also very convenient to have a range of test criteria, allowing testers to "start small" (with BC) and move up to stronger criteria (MBC and PW) when needed.

The strong documentation and automation of our tests also helped with a problem called *data aging*. In financial calculations, tests during one reporting cycle (for example, a month) have to change to be used in another reporting cycle. By designing our tests in an abstract way, the same abstract tests could be reused in multiple reporting cycles by instantiating them with new values. Not surprisingly, the same characteristics of the tests made it easy to regenerate new tests when requirements and design changed.

One disadvantage of input space partitioning is that the quality of the results depended somewhat on how well the testable functions are identified and how discrete they are. For example, system 3 initially considered all the calculators as one single testable function. When the 11 separate calculations were considered as individual testable functions, they become very simple and straightforward. ISP also has the potential to generate a lot of tests, so is not effective without strong automation. If not designed carefully, the pairwise criterion can lead to many invalid tests. Both of these problems were present with the tool used in this study, but not in more modern tools such as PICT (Czerwoka 2006) and ACTS (Kacker and Kuhn 2008).

Automating the requirements modeling approach provided many advantages, starting with the fact that the tool allowed tests to be quickly generated from the model. When modeled early, the requirements let the test analyst approximate the number of tests needed. The FTM tool also provides clear traceability from requirements to tests, as well as helping ensure tests are repeatable and detailed, important audit requirements for the testing. We were also able to share the requirements models, in their tree structure, with business analysts, programmers,

and testers, which greatly improved understanding of the entire process. Having the models available also made it very easy to adapt to changes in the requirements, and identify relations or constraints among input attributes to the software.

A disadvantage of the modeling approach is that it put a burden on the testers. To create the models, the test design team needs to understand software design and construction to do things like analyze UML diagrams and anticipate potential programming mistakes. In addition, the test team also needs to have substantial domain knowledge. We found that few people have both kinds of knowledge, so the teams must be well formed and have good communication. We also found that different test designers modeled the same requirements differently. Some designers wanted to refine the models continuously, seeking unachievable perfection, whereas others were quicker but made mistakes such as omitting important requirements or creating lots of redundant tests (as in system 2). Another problem encountered is that different teams have different development processes, causing management overhead in adapting the new testing ideas to each different process.

A problem we identified early is that Freddie Mac's software exhibits both low controllability and low observability. We interpret the high statement coverage to mean that we were able to solve the controllability problem. We addressed the observability problem by asking the programmers to log intermediate values; this made it much easier to diagnose the differences in expected and actual results.

References

Alluri C (2008) Testing calculation engines using input space partitioning and automation. Master's thesis, Department of Information and Software Engineering, George Mason University, Fairfax VA. Available on the web at: http://www.cs.gmu.edu/∼offutt/

Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press, Cambridge, UK. ISBN 0-52188-038-1

Ammann P, Offutt J, Huang H (2003) Coverage criteria for logical expressions. In: Proceedings of the 14th international symposium on software reliability engineering, Denver, CO, IEEE Computer Society Press, Los Alamitos, CA, pp 99–107

Bach J (2005) Allpairs test case generation tool. http://www.satisfice.com/tools.shtml. Accessed June 2012

Beizer B (1990) Software testing techniques, 2nd edn. Van Nostrand Reinhold, Inc, New York NY. ISBN 0-442-20672-0

Czerwoka J (2006) Pairwise testing in real world: practical extensions to test case generators. In: Proceedings of the 24th annual pacific Northwest software quality conference, Portland OR, USA, pp 419–430

Freedman RS (1991) Testability of software components. IEEE Trans Softw Eng 17(6):553–564

Grindal M, Offutt J, Andler SF (2005) Combination testing strategies: a survey. Softw Test Verif Reliab 15(2):97–133

Grindal M, Offutt J, Mellin J (2007) Conflict management when using combination strategies for software testing. In: Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, pp 255–264

IBM (2011) Rational robot. Online http://www-01.ibm.com/software/awdtools/tester/robot/. Accessed July 2011

Kacker R, Kuhn R (2008) Automated combinatorial testing for software-beyond pairwise testing. Online http://csrc.nist.gov/groups/SNS/acts/. Accessed June 2009

Lander J, Orphanides A, Douvogiannis M (1997) Earnings, forecasts and the predictability of stock returns: evidence from trading the s&p. J Portf Manage 23:24–35

Myers G (1979) The art of software testing. Wiley, New York, NY

Ostrand TJ, Balcer MJ (1988) The category-partition method for specifying and generating functional tests. Commun ACM 31(6):676–686
Ostrand TJ, Sigal R, Weyuker EJ (1986) Design for a tool to manage specification-based testing. In: Proceedings of the workshop on software testing, Banff, Alberta. IEEE Computer Society Press, Los Alamitos, CA, pp 41–50
Sortino F, Price L (1994) Performance measurement in a downside risk framework. J Invest 3 (3):59–64
Voas JM (1992) PIE: a dynamic failure-based technique. IEEE Trans Softw Eng 18(8):717–727

Jeff Offutt is Professor of Software Engineering in the Volgenau School of Information Technology at George Mason University. He has part-time visiting faculty positions at the University of Skovde, Sweden, and at Linkoping University, Sweden. His current research interests include software testing, analysis and testing of web applications, software evolution, and usable security. He has published over 145 refereed research papers in software engineering journals and conferences, and invented numerous test techniques, many of which are in widespread industrial use. Offutt is co-editor-in-chief of Wiley's journal of Software Testing, Verification and Reliability, co-founded the IEEE International Conference on Software Testing, Verification and Validation (ICST), was its first steering committe echair, and was Program Chair for ICST 2009. He is on the editorial boards for the Empirical Software Engineering Journal, the Journal of Software and Systems Modeling, and the Software Quality Journal, and was on the IEEE Transactions on Software Engineering from 2001 to 2005. He is co-author of the book Introduction to Software Testing. He received the Best Teacher Award from the Volgenau School in 2003 and was named a GMU Outstanding Faculty member in 2008 and 2009. Offutt received a PhD degree in Computer Science from the Georgia Institute of Technology, and is a member of the ACM and IEEE Computer Society. He has consulted with numerous companies on issues pertaining to software testing, usability, and software patents.

Biography and photo was not available for Chandra Alluri.