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 video where you record your presentation, please email a web link of your video to wasif.afzal@mdh.se.

The deadline for submission is 18-03-2017.

Instructions:

1. In this assignment, you will read two papers. These papers are as following (These papers are attached below with these instructions):
   - 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.
   - 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 video where you do a presentation (maximum of 10 slides) summarizing and reflecting on these papers. Think about answering the following aspects:
   - What are the goals/objectives/research questions addressed by each paper?
   - 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?
   - 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?
   - 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?
   - 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.

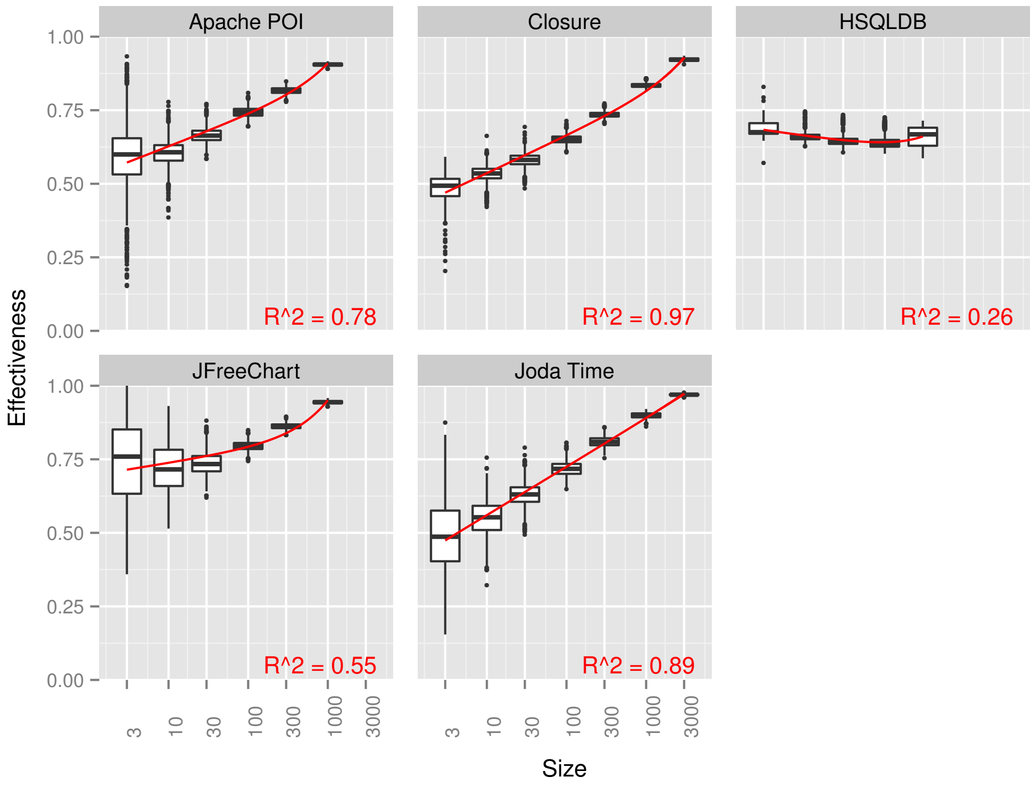


Figure 2: Normalized effectiveness scores plotted against size for all subjects. Each box represents the 1000 suites of a given size that were created from a given master suite.

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 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

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

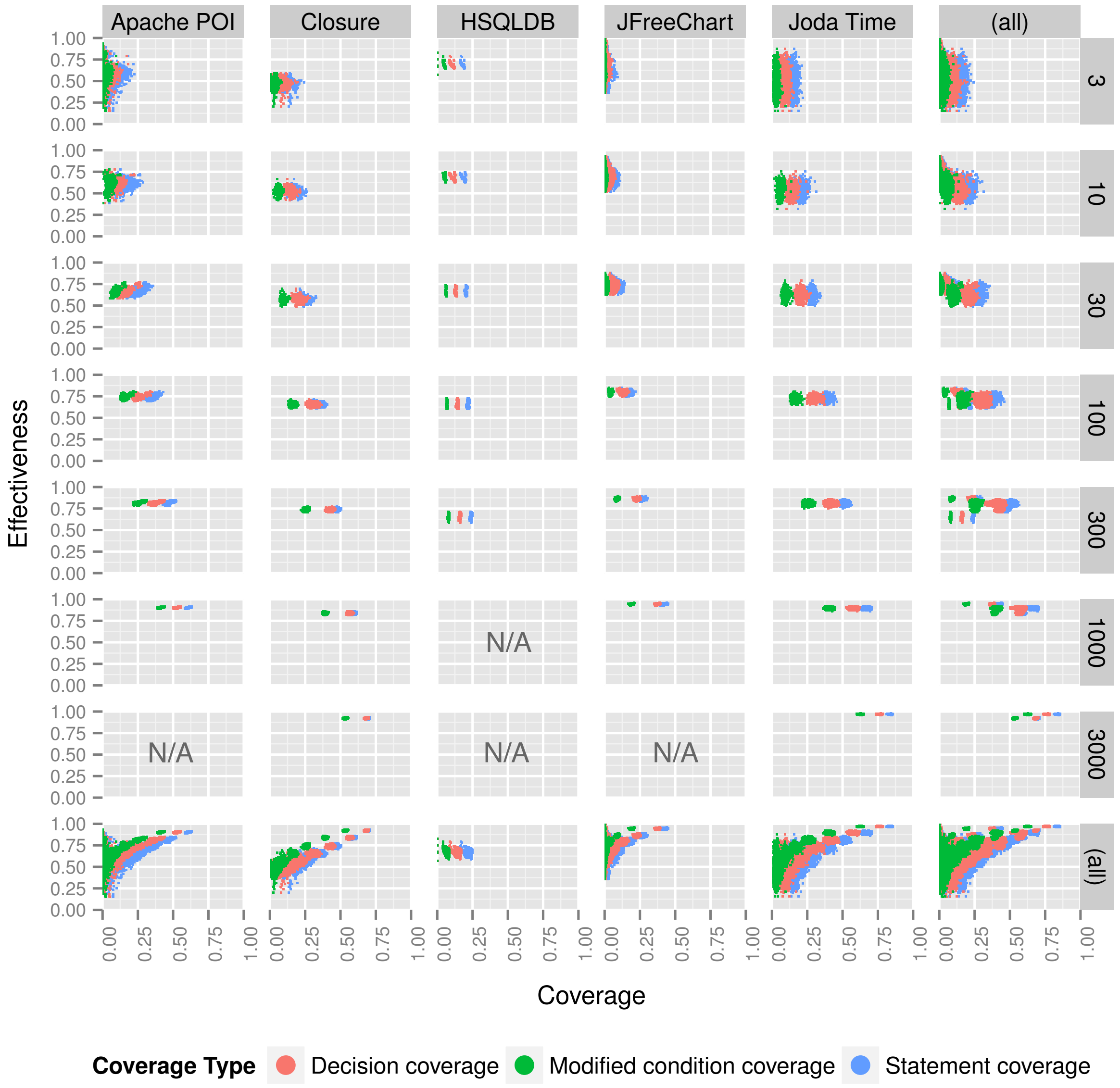


Figure 3: Normalized effectiveness scores (left axis) plotted against coverage (bottom axis) for all subjects. Rows show the results for one suite size; columns show the results for one project. N/A indicates that the project did not have enough test cases to fill in that frame.

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.

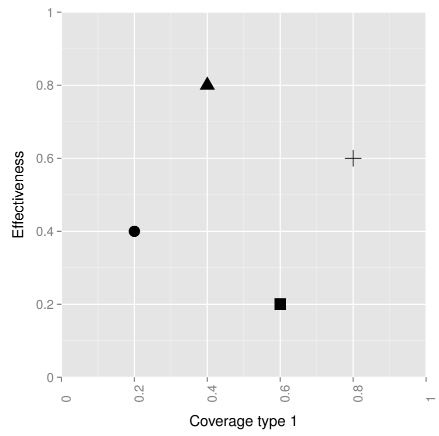
Apache POI fell at the other extreme. For this project, the correlation between coverage and the non-normalized effectiveness measurement was 0.94 when suite size was ignored, but dropped to a range of 0.46 to 0.85 when suite size was controlled for. While this is in some cases a large drop, a correlation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



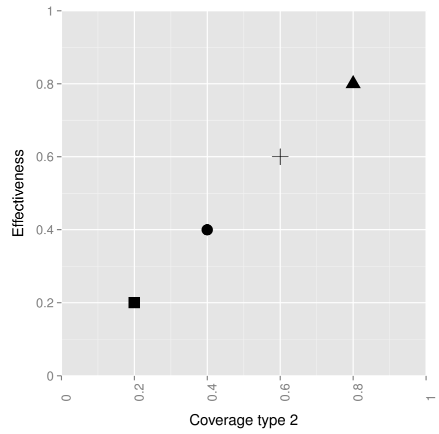


Figure 4: Hypothetical graphs of effectiveness against two coverage types for four test suites. The top graph shows a coverage type that is not correlated with effectiveness; the bottom graph shows a coverage type that is correlated with effectiveness.

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.

An Industrial Study of Applying Input Space Partitioning to Test Financial Calculation Engines

Jeff Offutt
Software Engineering
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

Chandra Alluri
Freddie Mac
McLean, VA, USA
chandra_alluri@freddiemac.com

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

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) [2,8] 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* [7] 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* [7] is also low. Software that exhibits low controllability and observability is notoriously hard to effectively evaluate during system testing [7]. (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 [16], or it may help estimate market direction, such as the Federal Reserve model [12].

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 [2, 7, 17]. Testability is largely influenced by two aspects of software, controllability and observability. Ammann and Offutt [2] 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. Offutt and Ammann [2] 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* [2,8]. 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 [14,15]. 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 [5], Myers [13] 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.

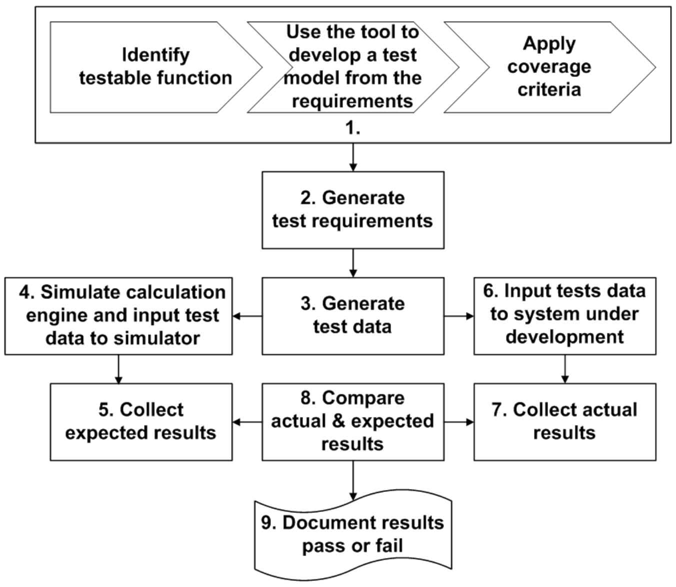

Identify testable function
Use the tool to develop a test model from the requirements
Apply coverage criteria
1.
2. Generate test requirements
3. Generate test data
4. Simulate calculation engine and input test data to simulator
6. Input tests data to system under development
5. Collect expected results
8. Compare actual & expected results
7. Collect actual results
9. Document results pass or fail

Fig. 1 Modeling process to test calculation engines

The test modeling process followed 10 steps, as adapted from Beizer [5].

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).
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 [3] 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* [5]. 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 [4]. 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 [1]. 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 calcula-

[1] We used Bach's PERL program to generate pairwise test requirements [4]. This is probably more tests than necessary and more modern tools, such as NIST's ACTS [11], 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

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

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*

tions. 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 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.

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*

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 [10]. The system has a feature called *import contracts* that allows all tests to be 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

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

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 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 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 [1].

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

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 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 [1].

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 [1].

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.

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	.27
MBC	82	17	.21
PW	302	23	.08
RM	158	7	.04

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 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 [11] 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.

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%						

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 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.

[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%						

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 [9] 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 as NIST's ACTS [11] 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 [2,8]. 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 [6] and ACTS [11].

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

1. C. Alluri. Testing calculation engines using input space partitioning and automation. Master's thesis, Department of Information and Software Engineering, George Mason University, Fairfax VA, 2008. Available on the web at: http://www.cs.gmu.edu/∼offutt/.
2. P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.
3. P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 99–107, Denver, CO, November 2003. IEEE Computer Society Press.
4. J. Bach. Allpairs test case generation tool. Online, 2005. http://www.satisfice.com/tools.shtml, last access June 2012.
5. B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
6. J. Czerwoka. Pairwise testing in real world: Practical extensions to test case generators. In *Proceedings of the 24th Annual Pacific Northwest Software Quality Conference*, pages 419–430, Portland OR, USA, October 2006.
7. R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.
8. M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, September 2005.
9. M. Grindal, J. Offutt, and J. Mellin. Conflict management when using combination strategies for software testing. In *Australian Software Engineering Conference (ASWEC 2007)*, pages 255–264, Melbourne, Australia, April 2007.

10. IBM. Rational robot. Online. http://www-01.ibm.com/software/awdtools/tester/robot/, last access July 2011.
11. R. Kacker and R. Kuhn. Automated combinatorial testing for software-beyond pairwise testing. Online, 2008. http://csrc.nist.gov/groups/SNS/acts/, last access June 2009.
12. J. Lander, A. Orphanides, and M. Douvogiannis. Earnings, forecasts and the predictability of stock returns: Evidence from trading the S&P. *Journal of Portfolio Management*, 23:24–35, 1997.
13. G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.
14. T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.
15. T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Proceedings of the Workshop on Software Testing*, pages 41–50, Banff, Alberta, July 1986. IEEE Computer Society Press.
16. F. Sortino and L. Price. Performance measurement in a downside risk framework. *The Journal of Investing*, 3(3):59–64, Fall 1994.
17. J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8), August 1992.