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

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

Jeff Offutt · Chandra Alluri

Published online: 23 September 2012

Editor: James Miller

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

Keywords Software testing · Industrial study · Input space partitioning

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

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

1 Introduction

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

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

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

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

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

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

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

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

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

2 Characteristics of Calculation Engines

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

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

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

2.1 Specification Formats for Calculation Engines

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

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

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

2.2 Characteristics of Design and Implementation of Calculation Engines

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

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

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

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

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

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

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

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

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

3 Test Approach

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

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

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

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

3.1 Input Space Partitioning

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Requirements Modeling

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

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

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

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

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

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

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

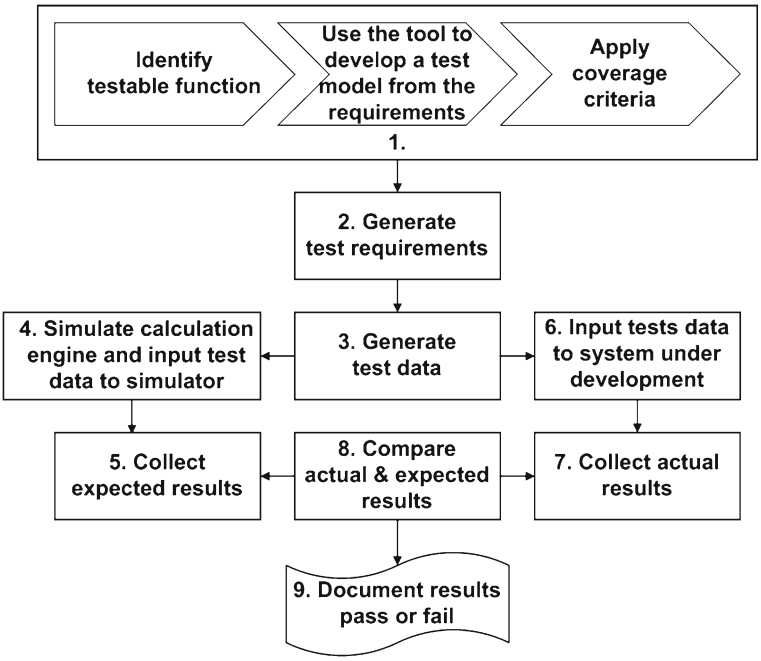

Fig. 1 Modeling process to test calculation engines

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

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

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

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

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

3.3 The Fusion Test Modeler

FTM was developed to meet seven essential needs.

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

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

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

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

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

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

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

4 Software Systems Studied

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

4.1 Contract Pricing

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

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

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

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

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

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

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

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

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

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

Table 1 Contract partitions and blocks

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

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

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

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

Table 2 Contract pricing partitions and blocks

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

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

4.2 Loan Pricing

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

Table 3 Contract pricing stage 2 base choice tests

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

Table 4 Contract pricing stage 2 multiple base choice tests

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

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

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

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

Table 5 Contract pricing stage 2 pairwise tests

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

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

4.3 Amortization

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

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

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

Table 6 Contract pricing stage 2 requirements modeling tests

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

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

4.4 Static Effective Yield

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

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

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

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

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

5 Results

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

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

5.1 Fault Detection

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

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

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

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

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

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

Table 8 Fault efficiency – all four studies

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

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

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

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

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

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

5.2 Coverage Measurement

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

Table 9 Statement coverage results

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

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

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

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

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

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

5.3 Observations

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

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

Table 10 Functional requirements coverage results

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

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

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

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

5.4 Threats to Validity

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

6 Conclusions and Future Work

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

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

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

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

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Biography and photo was not available for Chandra Alluri.