Testing Across Configurations: Implications for Combinatorial Testing

Myra B. Cohen, Joshua Snyder, Gregg Rothermel
Dept. of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{myra,jsnyde,grother}@cse.unl.edu

ABSTRACT

User configurable software systems allow users to customize functionality at run time. In essence, each such system consists of a family of potentially thousands or millions of program instantiations. Testing methods cannot test all of these configurations, therefore some sampling mechanism must be applied. A common approach to providing such a mechanism has been to use combinatorial interaction testing. To date, however, little work has been done to quantify the effects of different configurations on a test suites' operation and effectiveness. In this paper we present a case study that investigates the effects of changing configurations on two types of test suites. Our results show that test coverage and fault detection effectiveness do not vary much across configurations for entire test suites; however, for individual test cases and certain types of faults, configurations matter.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Measurement,Verification

Keywords

Combinatorial interaction testing, Empirical study, Code coverage, Configurable software

1. INTRODUCTION

User configurable software systems provide late binding of system functionality, allowing combinations of program features to be selected at run time by the user rather than at build time by the system developer. For instance, a web browser may allow the user to turn the Java Script feature on and off at run-time, or provide a series of security features that can be selected based on the user's preferences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
A-MOST '06, November 7, 2006, Raleigh, North Carolina, USA.
Copyright 2006 ACM 1-58113-000-0/00/0004 ...$5.00.

A challenge for testing user configurable systems is that a single such system can support an enormous set (thousands, or even millions) of configurations. Each configuration may behave differently under the same software test sequences, with different code bases executed, and different sets of faults uncovered [17, 21].

For example, consider the Firefox browser for Mac OS X. The browser has many possible user defined configurations. Under the "Advanced Tab", the user may tune five options for "Tabbed Browsing"; the first option ("Open Links In") has three possible values ("New Window", "New Tab", "Most Recent"), while the others each have two (see Table 1). The user can create 48 configurations for just this tab, and this is only a small portion of the full configuration space. As a second example, in a recent version of Internet Explorer for Windows XP there are 31 configurable options on the security tab. Of these, 10 are binary, 19 have three settings and two have four settings. As a result there are $2^{10} \times 3^{19} \times 4^2 = 19,042,491,875,328$ configurations. Again, this is only a snapshot of the entire configuration space.

	Open Links In	Tab Bar	Select New Tabs from Links	Select New Tabs from history	When Closing Multiple Tabs
value	New Window	Hidden	Yes	Yes	Warn
	New Tab	Visible	No	No	Don't Warn
	Most Recent				

Table 1: User Configurations

Assume that the Firefox browser has been tested with the "Open Links in New Window" and "Warn When Closing Multiple Tabs" options selected. Even if this configuration passes all tests it is possible that, were the first option changed to "Open Links in Most Recent", one of these tests might fail. This failure may be caused by initialization code, or a peculiar program state set under one option yet not reached under the other. The type of fault causing this failure is called an *interaction fault*.

Although in theory, tests could be run under all possible configurations in order to detect *interaction faults*, in practice this is infeasible. A test suite for *each* configuration of Firefox may include hundreds or thousands of test cases. If four hours are needed to run the entire test suite, the complete set of 48 configurations would require eight days to test. Suppose there were ten configurable options, each with five possible values; in this case testing would require $5^{10} = 9,765,625 \times 4$ hours (4,459 years). Alternatively, instead of testing all configurations of the Firefox browser, sampling heuristics can be used to select subsets of the configuration space. For instance, one might randomly select configurations, or test only "supported" configurations.

Input testing (testing aimed at selecting values from a system's input space) suffers from a similar "combinatorial explosion" problem. One method that has been used to sample an input space is to include all *pairs* or t-way combinations of inputs, where t is a defined *strength* of testing [2, 4, 5, 6, 7, 8, 9, 16]. Empirical evidence suggests that this *combinatorial interaction testing* approach can be effective [1, 4, 7, 9].

Applying this same concept to the Firefox example, one could test *all pairs* of settings using only six configurations (see Table 2). In this set of configurations all settings of "Tab Bar" are tested with all settings for "Select New Tabs from Links", all settings for "Select New Tabs from History", and all settings for "When Closing Multiple Tabs". In addition, all settings for "Select New Tabs from Link" are tested with "Select New Tabs from History" and "When Closing Multiple Tabs".

System Configuration	Open Links In	Tab Bar	Select New Tabs from Links	Select New Tabs from history	When Closing Multiple Tabs
1	New Window	Hidden	Yes	Yes	Don't Warn
2	New Window	Visible	No	No	Warn
3	New Tab	Visible	No	Yes	Don't Warn
4	Most Recent	Hidden	No	Yes	Warn
5	New Tab	Hidden	Yes	No	Warn
6	Most Recent	Visible	Yes	No	Don't Warn

Table 2: Testing 2-Way Interactions

The potential for using combinatorial interaction testing cost-effectively on configurable systems has been suggested in several studies: a retrospective examination of flaws in the Mozilla web browser and an Apache database determined that 71% and 58%, respectively, of the faults in these systems were caused by interactions of two or more configuration settings [14, 15]; a distributed CORBA middleware was effectively sampled using combinatorial testing for fault characterization [22, 23].

The majority of the literature on the cost-effectiveness of combinatorial interaction testing, however, has focused on input testing [2, 3, 4, 6] in which, as described above, the inputs to the system are partitioned into equivalence classes. Pairs (or t-tuples) of combinations of classes define the individual test cases. In this context, a combinatorial interaction test suite is a set of test cases that define all the specified combinations of input parameters. In contrast, on configurable systems, a combinatorial interaction test suite defines the set of configurations under which each test case in a given test suite is run.

The issues involved in combinatorial testing of configurations have not yet been carefully studied. For instance, it is not clear how sensitive test suites and individual test cases are to changing configurations. In addition, the current approach of testing a balanced set of interactions across the whole application may not be the most effective approach. Finally, there may be certain types of faults or test suites that will benefit more from these methods. In order for combinatorial interaction testing to be used most effectively for user configurable systems, it is important to understand the implications of changing configurations on the effectiveness of software test suites.

To provide further understanding of these issues, we have designed and performed a case study aimed at quantifying the effects of changing configurations on two types of test suites applied to an open source web browser. The first test suite is a specification-based GUI test suite, and the second is a set of random fuzz tests [11]. We examine the effectiveness of each test suite across three different configuration spaces of the browser. We utilize several metrics to understand the changes across the configurations, focusing on the effects of configurations on code and fault coverage.

Our results show that test coverage and fault detection effectiveness do not vary much across configurations for entire test suites; however, for individual test cases and certain types of faults the configuration tested matters.

The rest of this paper is organized as follows. Section 2 describes the study, Section 3 presents and analyzes the data, and Section 4 discusses the implications of the results. Section 5 summarizes related work, and Section 6 concludes.

2. EMPIRICAL STUDY

Our study is designed to quantify the effectiveness of testing a user configurable system, given a fixed test suite, across changing configurations. In particular, we wish to address the following research questions:

RQ1: How does changing the configuration of a software system affect fault finding for a given test suite?

RQ2: How does changing the configuration of a software system affect overall code coverage?

RQ3: Does the granularity of our measure (i.e test suite or test case) affect these results?

The rest of this section describes our study object, variables and measures, methodology, and threats to validity.

2.1 Object of Analysis

As an object of analysis for this study we used the open source web browser, MyIE [24]. MyIE is a wrapper for the Internet Explorer engine. It offers several special features, including support for up to six IEs each in its own tab, user definable search engines, multiple search engine support, favorites support, visual bookmarks, grouped bookmarks, and online version check support. The primary reason we chose this program for our study is that it is highly configurable. Not only can one alter the configurations included with Internet Explorer, but MyIE has an entire set of configurations specific to itself. In addition, we have access to a version of MyIE that, for a previous study by other researchers [13], has been provided with several other artifacts required for our study.

The version of MyIE we utilized was obtained originally from sourceforge.net. It has approximately 41 KLOC, contained in 64 classes and 2793 basic blocks. The code was instrumented previously to capture basic block coverage, and seeded with 77 faults, contained in 51 unique blocks. (The authors of this paper were not involved in the prior fault seeding process.) As the modified MyIE program runs, coverage packets are sent to a database. These packets contain an ordered list of the blocks traversed for each test case under each configuration.

We used two different test suites for our study. The first test suite, a specification based test suite, is a subset of the test suite originally developed for MyIE by the prior researchers. The original test suite included 243 test cases that exercise each menu option one time; our test suite initially contained the 145 of these test cases that execute the most common menu options. Of these test cases, we were forced to eliminate 10 during the study because they failed to execute to completion under certain configurations. We used a commercial GUI playback tool, Vermont HighTest Plus[1] to execute these test cases.

The second test suite that we utilized is a random test suite used to uncover security faults such as buffer overruns. *Fuzz* testing [11, 18] is a testing method used to detect security flaws in software. In fuzz testing, random messages are sent to a program's message

[1] http://www.vtsoft.com

MyIE Option	Config One (n=32)	Config Two (n=72)	Config Three (n=300)	Default Config
Tab Position	Bottom, Top	Top	Top	Top
Multiple Line Tab	On, Off	On	On	On
Auto Adjusting Tab	On, Off	On	On	On
Display New Tab Next to Current	On, Off	Off	Off	Off
Right Click and hold Ctrl	None, Close window	Close window, Auto save	None	None
Max Tab Width	72	0, 72, 33000	72	72
On Startup	Resume last visit	Blank window, Resume last visit, Open most favorites, Open folder in favorites	Resume last visit	Resume last visit
Max Tab Characters	11	0, 72, 33000	11	11
Auto Save	Whole page html	Whole page html	Whole page html, Whole page mht (archive), Original page html, Quick save, Web page html	Whole page html
Allow Scripts	Off	Off	On,Off	Off
Left Click Tab	Close window	Close window	Off, Close window, Save as, Save html, Auto save	Close window
Default New Window	Blank	Blank	Blank, Home, Current	Blank
Use Mouse Gestures	On	On	On, Off	On

Table 3: Configuration variables and corresponding values

queue. We modified the Windows NT fuzz tool [11] for our study. The tool attaches to a specified Windows process (in this case the main MyIE window) and sends the specified number of random Win32 messages, and for our study, we consider each run of this tool a test case. Each fuzz test thus created consists of a number of random "Send Messages" sent to the running program. We created a pool of fuzz test cases, each containing between 100 and 5000 messages. The number of messages sent for each test case was also selected randomly. Our initial fuzz test suite contained 50 test cases. We then eliminated test cases that caused MyIE to stop responding, (e.g. by finding non-seeded faults) because these prevented us from collecting the required coverage data. This left us with 35 fuzz test cases in the fuzz test suite.

2.2 Variables and Measures

The independent variables in this study are the different MyIE configurations and are explained in Section 2.2.1. The dependent variables in this study are a set of five metrics that analyze the acquired data from different perspectives, and are described in Section 2.2.2.

2.2.1 Independent Variables

We selected three sample configuration spaces of MyIE to manipulate. Since our goal was to manipulate all of the options selected for each configuration space selected each time, it was infeasible to use the entire set of options for the MyIE browser. (In total, the MyIE wrapper has 29 binary options, two ternary options, four options with five values, one option with seven values, and 17 text based options that must be partitioned into equivalence classes to test all combinations. If we consider just the fixed size options there are $2^{29} \times 3^2 \times 5^4 \times 7 = 21,139,292,160,000$ feasible configurations of this browser.) Thus, instead, we chose three sample spaces. Table 3 shows the MyIE options that were altered in at least one of the three configuration spaces, their values in each space, and the default values used when an option is not being manipulated in a configuration space.

For the first configuration space we selected five options from the Tab panel of MyIE. For each option, we used two of the possible values resulting in a configuration space with 32 configurations. For the second configuration space we manipulated four options from the Tab panel; these options had two, three, four and three possible values, respectively. This gives us a configuration space of size 72. Finally, in the third configuration space we chose to manipulate five options across panels. We selected one from each of the "Tab", "Save", "Window", "General" and "Download" panels. This configuration space contains 300 options.

2.2.2 Dependent Variables

Since our goal is to investigate the changes to fault detection effectiveness and code coverage across configurations we need metrics that can quantify these changes. We also wish to quantify effects at different levels of granularity (e.g., test suite versus test case). We therefore selected dependent variables based on several defined by Elbaum et al. [10].

In [10], code coverage was quantified for components across system releases. We modify each of the metrics used there to consider code coverage and fault detection across configurations. Elbaum et al. use a component matrix in which rows represent blocks and columns represent test cases. In our work we wish to examine both overall block coverage of the test suites as well as fault coverage. Where fault coverage is concerned, ideally, we could hope to use an oracle to determine whether faults are actually detected by test cases. With the MyIE object, however, a deterministic oracle is not available, and a non-deterministic oracle threatens the validity of conclusions that might be drawn about fault detection across runs on different configurations. Thus, we approximate fault detection

1	0	0	0	0	1
2	1	1	0	0	0
3	0	1	1	0	1
4	0	0	0	0	0
5	1	0	1	0	1
6	0	0	0	0	0
7	0	0	0	0	0

B_1

0	0	0	0	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

F_1

0	1	0	0	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

F_2

Figure 1: Block and Fault Matrices

by considering fault coverage, which we obtain by examining only the subset of blocks containing seeded faults covered during the study.

We thus define two matrices that can be used as the basis for all of our metrics. One contains all of the block information, the second contains just seeded fault blocks. Let B be a $b \times t$ matrix, where b is the number of unique blocks in the program, and t is the number of test cases in the test suite. If cell (i, j) in B contains a 1, this means that test case j traversed block i, otherwise the cell contains 0. We call B a *block matrix*. Let F be an $f \times t$ matrix, where f is the number of unique blocks containing faults and t is defined as as previously. If cell (i, j) in F contains a 1, this means that test case j traversed a block containing fault i. We call F a *fault matrix*. Figure 1 shows examples of a block matrix, B_1, and two fault matrices, F_1 and F_2. In this example, there are seeded faults in blocks $1, 2, 6,$ and 7 of B_1, and thus F_1 is the corresponding fault matrix.

The block matrices for the specification based test suites used in our study contain 2793 rows and 135 columns (one for each valid test case). The fuzz block matrices contain 2793 rows and 35 columns. The fault matrices for our specification test suites contain 51 rows and 135 columns and the fuzz fault matrices contain 51 rows and 35 columns.

We use these two types of matrices to calculate each of several dependent variables: block coverage (BC), matrix density (MD), fault coverage (FC), coverage across faults (CAF) and coverage across tests (CAT), as follows.

Block Coverage

BC measures the percentage of blocks covered by a given test suite. For each row, i, of block matrix B, $block_count_i = 1$ if at least one column contains a 1 (i.e. any test covers this block) and 0 otherwise. Then:

$$BC = \frac{\sum_{i=1}^{b}(block_count_i)}{b} \times 100$$

In Figure 1, the BC for matrix B_1=57.14%.

Fault Coverage

FC measures the percentage of faults covered by a given test suite. For each row, i, of fault matrix F, $fault_count_i = 1$ if at least one column contains a 1, (i.e. any test covers this fault block) and 0 otherwise. Then:

$$FC = \frac{\sum_{i=1}^{f} fault_count_i}{f} \times 100$$

Matrices F_1 and F_2 of Figure 1 have FC values of 50% and 25%, respectively.

Matrix Density

MD measures the percentage of the fault blocks covered by a given test suite. This metric sums all of the 1's in a fault matrix F, and divides by the total number of cells:

$$MD = \frac{\sum_{i=1}^{f}\sum_{j=1}^{t} F[i][j]}{f \times t} \times 100$$

Matrices F_1 and F_2 of Figure 1 have MD values of 15% and 10%, respectively.

Coverage Across Faults

CAF compares a pair of fault matrices, F_1 and F_2, and determines the number of rows that differ between them. If rows $F_1 i$ and $F_2 i$ have at least one location (i, j) that differs, $diff_Fcount_i = 1$. Then:

$$CAF = \frac{\sum_{i=1}^{f} diff_Fcount_i}{f} \times 100$$

The CAF value for matrices F_1 and F_2 in Figure 1 is 50%. In our study, CAF is calculated for all $\binom{n}{2}$ pairs of configurations where n is the number of configurations in a configuration space.

Coverage Across Tests

CAT also compares two fault matrices, F_1 and F_2, and measures the percentage of columns that differ between them. Let $diff_Tcount$=1 if any row i for column j in matrix F_2 differs from F_1. Then:

$$CAT = \frac{\sum_{i=1}^{t} diff_Tcount_i}{t} \times 100.$$

The CAT value for matrices F_1 and F_2 in Figure 1 is 40%. In our study we examined all $\binom{n}{2}$ pairs of configurations.

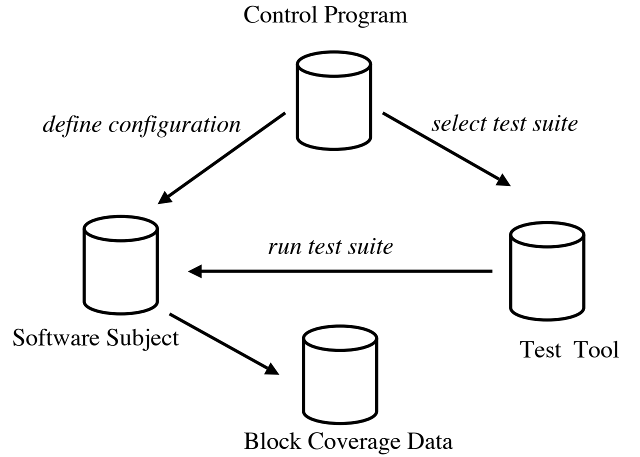

Figure 2: Study Framework

2.3 Study Methodology

All of our data was collected on a 1.6 GHz laptop running Windows XP professional, a MySQL server, and a local Apache Web server. We used an overall control program to modify the MyIE configurations, run the testing tools and collect fault coverage data. For each test suite and configuration space, we iterated through all of the configurations and ran all of the test cases. The MyIE configurations are defined by a text file that we regenerated for each configuration at the start of the run. Cleaning tasks were performed after each test case completed. The framework for our study is shown in Figure 2. In this figure, the control program consists of a script that modifies the MyIE configuration, performs pre and post testing clean up tasks, and starts the testing tool. The test tool used

	Min	Max	Mean	SDev
Sp_1	31.37	33.33	32.35	1.00
Sp_2	**29.41**	**33.33**	31.54	1.26
Sp_3	33.33	33.33	33.33	NA
Fz_1	19.61	19.61	19.61	NA
Fz_2	19.61	19.61	19.61	NA
Fz_3	19.61	19.61	19.61	NA

Table 4: FC Value Statistics

is either Vermont HighTest or the fuzz tool. As the test suite ran, packages containing the block coverage were sent to a server where they were processed to create the block and fault matrices.

2.4 Threats to Validity

Where external validity is concerned, our study involves only a single software system with three configuration spaces and two specific test suites, and with seeded faults; studies on additional artifacts will be needed to determine the degree to which results may generalize. However, our object of study is representative of an important class of user-configurable systems, and its associated artifacts (tests, faults) were not created by us, eliminating one potential source of bias.

With respect to internal validity, we required several analysis tools for this study, and errors in such tools might bias our results. To reduce this possibility we validated our tools on subsets of our data. Also, we were forced to remove test cases that were not able to execute under all configurations, and fuzz test cases on which the system did not respond. Had these test cases been executable they might have exhibited different results.

Where construct validity is concerned, code coverage and fault detection are just two measures of test suite value; other measures such as the cost of testing and the robustness of tests across configurations as systems evolve may also matter. Also, our measure of fault detection effectiveness relies on fault coverage data, and thus may overstate fault detection capabilities, since in practice some faults might be covered but not propagate their effects to output.

3. DATA AND ANALYSIS

We now present and analyze the data gathered in our study. We organize the discussion in terms of our three research questions.

3.1 Fault Finding Across Configurations

To address RQ1 we focus on the overall fault finding capabilities of our test suites across configurations, using the fault coverage (FC) and matrix density (MD) metrics. We present results for both of the test suites. In our discussion, we denote specification based test suites by labels Sp_1, Sp_2 and Sp_3, and fuzz test suites by labels Fz_1, Fz_2 and Fz_3; here, the subscripts indicate configuration spaces.

The fault coverage metric results are depicted in Figure 3 and Table 4. For the specification based test suites, FC values ranged from 29.41% to 33.33% across all configuration spaces and configurations. The third configuration space yielded the best fault finding ability across configurations, while the second configuration space was the most configuration dependent. The maximum FC for the second configuration space matched that of the other two configuration spaces, but the median and mean FC values for this configuration space were lowest. In contrast, the fuzz test suite was insensitive, in terms of FC values, to differences in configuration spaces; these values remained fixed at 19.61%.

The matrix density (MD) metric provides a second view of the fault finding results across configurations (Figure 4 and Table 5). A denser fault coverage matrix (higher MD) indicates that more individual test cases covered a particular fault. MD appears relatively stable across configurations, for all configuration spaces and both types of test suites. For the specification based test suite, there was a less than one percent range in MD values across configurations (15.89 % - 16.78%). For the fuzz test suite, the first and third configuration spaces yielded no variance in MD values, while the second configuration yielded less than a two percent range (11.88% - 13.33%).

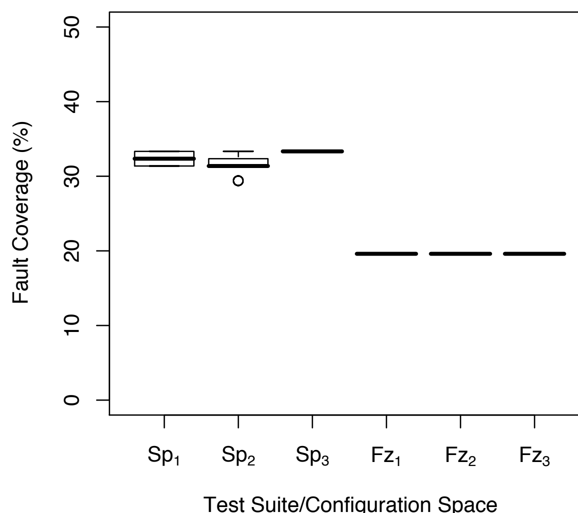

Figure 3: Fault coverage (FC) values per test suite and configuration space.

Since our data did reveal some differences in FC and MD across configurations, we analyzed the differences further by considering specific faults and the configuration settings that found and/or missed them. For the specification based test suite, between 16 and 17 of the 51 faults were found across configurations in configuration space 1, between 15 and 17 faults were found across configurations in configuration space 2, and the same 17 faults were found in each configuration in configuration space 3. Detection of two specific faults, faults 2 and 23, accounted for all of the variation seen in the first two configuration spaces. In Sp_1 fault 2 was found in one half of the configurations, but it was missed in the others. In Sp_2 this fault was found in only 24 of the 72 configurations, and fault 23 was found 54 times and missed 18 times.

We examined the configurations in which fault 2 was revealed in each of the three configuration spaces and compared the underlying configuration settings. In configuration space 1, finding fault 2 is associated with a single option, “Tab Position”. When this option is set to its default setting, “set to top,” the fault is found. When the option is set to “set to bottom”, the fault is missed. In configuration space 2, finding fault 2 is associated with a different option, “Maximum Tab Width”. When this option is set to 72 the fault is found. What is interesting to note is that a value of 72 for this option is used as the default option and not manipulated in configuration spaces 1 or 3. Because fault 2 is sometimes missed in configuration 1, the “Maximum Tab Width” option alone can’t explain these findings. It appears, instead, that all configurations with the combination of Tab Position “set to top” and Maximum Tab Width = 72 find this fault, while all others miss it.

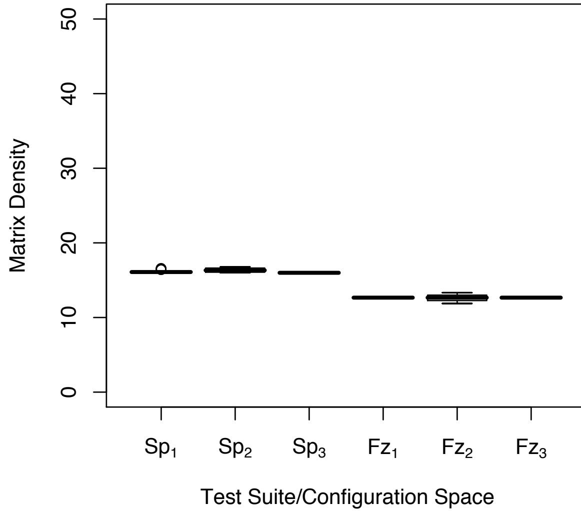

Figure 4: Matrix density (MD) values per test suite and configuration space.

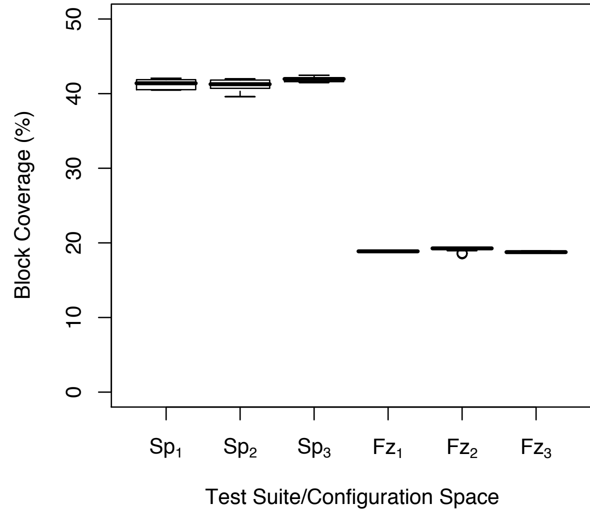

Figure 5: Block coverage (BC) values per test suite and configuration space.

	Min	Max	Mean	SDev
Sp_1	16.02	16.59	16.12	0.12
Sp_2	15.99	**16.78**	16.34	0.27
Sp_3	**15.89**	16.05	15.98	0.5
Fz_1	12.66	12.66	12.66	NA
Fz_2	**11.88**	**13.33**	12.63	0.52
Fz_3	12.66	12.66	12.66	NA

Table 5: MD Value Statistics

	Min	Max	Mean	SDev
Sp_1	40.49	42.07	41.24	0.69
Sp_2	**39.60**	41.96	41.16	0.71
Sp_3	41.50	**42.46**	41.85	0.23
Fz_1	18.76	19.01	18.87	0.09
Fz_2	**18.55**	**19.41**	19.21	0.22
Fz_3	18.76	18.90	18.82	0.07

Table 6: BC Value Statistics

The fuzz test suites found exactly ten faults in each configuration, for each configuration space. Of these, nine faults match those found by the specification based test suites, while one new fault was found. The two "configuration dependent faults," numbers 2 and 23, were never traversed by the fuzz test suite in any configuration.

3.2 Code Coverage Across Configurations

To address RQ2 we use the block coverage metric (BC). Figure 5 and Table 6 show the BC values for each test suite and configuration space. In total, 2793 blocks were instrumented in MyIE. Of these blocks, the specification based tests covered between 39.6 % and 42.6% across all configuration spaces and configurations – just a 3% difference. Configuration space 3 exhibited slightly higher overall block coverage than the other configuration spaces, while configuration space 2 exhibited the lowest.

The fuzz test suite covered approximately half as many blocks as the specification based test suite. The range across configurations for this test suite was less than one percent (18.55% - 19.41%).

3.3 Granularity of Measures

Our third research question focuses on the granularity of our measures. To address this question we use the two metrics, change across faults (CAF), and change across tests (CAT), that distinguish behaviors relative to individual faults and tests.

Figure 6 and Table 7 present CAF metric results. Recall that CAF tracks changes across configurations at the fault level. The data shows that there are many configurations that matched others exactly (e.g. have a CAF of 0.0) in terms of patterns of fault finding, while on average, between 4% and 8% of the configurations differed from other configurations. For the specification based test suite, configuration spaces 1 and 2 exhibited the same minimum and maximum CAF values, although a larger number of configurations differed in configuration space 2. The median CAF values for configuration spaces 2 and 3 were equal, and slightly higher than that for configuration space 1. The fuzz test suite behaved differently. For this suite, configuration spaces 1 and 3 exhibited no differences at all, whereas configuration space 2 exhibited a range of 13.73% and had the highest median and average CAF values.

The final metric, change across tests or CAT, detects changes across configurations at the test level. Figure 7 and Table 8 present the CAT data. The CAT metric exhibited large differences in this study, ranging from 4% to 20%. For the specification based tests, the second configuration space appears to exhibit the greatest range in CAT, and the highest mean and median, while the third configuration space exhibits the lowest. In the first and third configuration spaces for the fuzz test suite, the metric is 0.0, i.e., all configurations match exactly. In the second configuration space, however, the range in CAT is 11.43. The mean CAT for this configuration space is 4.35%.

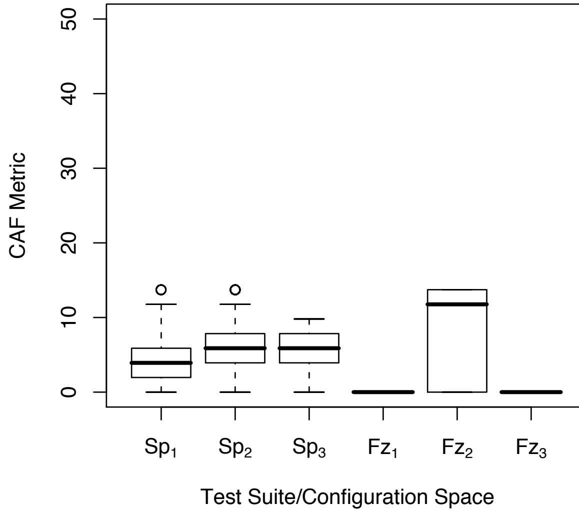

Figure 6: Change across faults (CAF) values per test suite and configuration space.

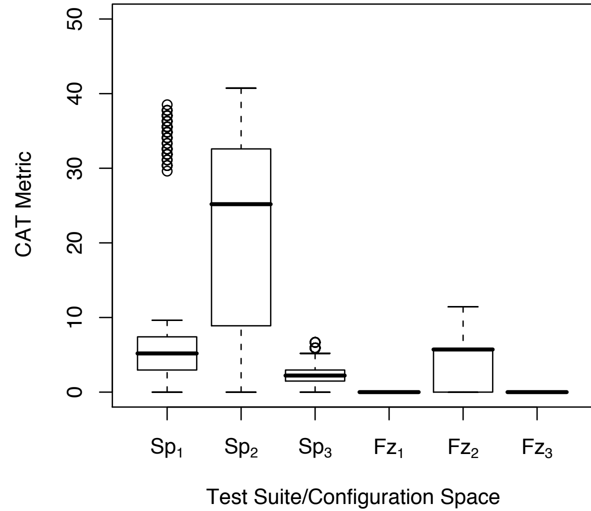

Figure 7: Change across tests (CAT) values per test suite and configuration space.

	Min	Max	Mean	SDev
Sp_1	**0.00**	**13.73**	4.66	2.86
Sp_2	0.00	13.73	6.48	2.90
Sp_3	0.00	9.81	5.41	2.31
Fz_1	**0.00**	0.00	0.00	NA
Fz_2	0.00	**13.73**	8.20	6.28
Fz_3	0.00	0.00	0.00	NA

Table 7: CAF Value Statistics

	Min	Max	Mean	SDev
Sp_1	**0.00**	38.52	9.80	11.57
Sp_2	0.00	**40.74**	20.67	12.50
Sp_3	0.00	6.67	2.38	1.25
Fz_1	**0.00**	0.00	0.00	NA
Fz_2	0.00	**11.43**	4.35	3.78
Fz_3	0.00	0.00	0.00	NA

Table 8: CAT Value Statistics

4. DISCUSSION

The results of this study show that although there are some differences in fault detection effectiveness and overall code coverage across configurations, they are relatively small. The granularity of our measure does matter, however, when considering results. At the test suite level there is less variation than at the test case level. We also observed that each of the configuration spaces utilized yielded slightly different results, suggesting that the specific combinations of options selected when manipulating a configuration space may be important. We now discuss each research question, and the implications for combinatorial interaction testing, in more detail.

4.1 RQ1: Test Suite Effectiveness

The FC and MD metrics reveal only very small differences between configurations, where fault detection is concerned. For the specification based test suite, the difference was the greatest, but still small. The fact that we saw slightly different results between different configurations suggests that the model and configuration space selected matters. For the fuzz test suite, there were no differences at all for two of the three configuration spaces. This may be explained by the fact that the fuzz based test suites are not designed for functional or code coverage but rather aim to find faults caused by exceptional conditions.

Furthermore, we saw that although some faults are configuration dependent, the majority are not, and are not impacted by the choice of configurations at the test suite level. In the specification based test suite only two of the 17 faults detected had a clear dependency on configuration.

4.2 RQ2: Code Coverage

The difference in code coverage across configurations was slightly higher for the specification based test suite than for the fuzz test suite. This may be due to the fact that the overall code coverage was twice as high for the specification based test suite. In general it does not appear that at the test suite level there are large differences in code coverage; however, even for the fuzz test suite, on which no differences were seen in the fault matrices for two configuration spaces, there were some slight differences in code coverage. Additional study will be needed to determine whether these results generalize to more coverage-complete test suites.

4.3 RQ3: Granularity of Measure

The choice of analysis at the test suite or test case level appears to have the greatest impact on our results. Both the CAF and CAT metrics showed the largest range of values. This was as high as 41% for CAT on the specification based test suite in configuration space 2, and 14% for CAF in configuration space 1 and 2. Even the fuzz test suite, which appears to be insensitive to configurations, displayed some differences in configuration space 2 (14% for CAF and 11% for CAT).

4.4 Combinatorial Testing: Implications

Only a small subset of the faults our study considered are configuration dependent; however, for these faults, the exact configuration used during testing did matter, and combinatorial interaction testing techniques such as pairwise testing may prove useful. We saw this phenomenon in fault number 2, where we were able to trace the success of its detection to a pair of option settings.

In general it does not seem that using combinatorial interaction techniques will greatly improve code coverage at the block level; still, if increased coverage is crucial, testing multiple configurations might offer a slight benefit.

The testing scenario that may benefit the most from testing across configurations is one in which only a small set of test cases are used. In particular, this may occur in regression testing situations where test suite subsets are utilized, e.g. through regression test selection [20]. In these situations, the impact of changing configurations may have a larger effect.

5. RELATED WORK

There has been a large body of work on methods for reducing the combinatorial interaction space of programs [1, 4, 12, 16, 19]. We do not summarize all of this work here, but we discuss some work in this area that uses Design of Experiments to select a subset of inputs or configurations. We focus on two closely related studies. In [4] D. Cohen et al. describe the automatic efficient test generator, AETG [4], that provides t-way interaction coverage between program inputs. They describe a study in which they capture code coverage for the Unix utility program *sort*. Code coverage metrics are presented for several models of the configuration space, but the authors do not examine all possible configurations, rather they focus only on the sample selected. In addition, they are not concerned with the entire configuration definition, but rather, use the sample selected to define the actual test cases.

In further work, Dunietz et al. examine code coverage for various strengths of interaction testing, i.e. 2-way, 3-way, etc. Again the focus is on the sample of the configuration test suite, rather than on using the entire set of configurations. In [23], Yilmaz et al. manipulate and test a configuration space with over 18,000 configurations for a distributed middleware system. They compare their results with samples based on the AETG methodology. Like us, they concern themselves with the entire configuration definition, however, their focus is on fault localization. They do not relate their findings to code coverage, nor do they compare results across multiple configuration models.

Our work differs from these prior studies in that we have considered several configuration spaces within a single software system, and for each, we have examined the entire set of configurations under two different test suites, and then related this to the code and fault coverage of each.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the results of a case study to examine the effects of changing configurations on a user configurable application, MyIE. We defined a set of metrics to determine the effectiveness of tests suites and test cases across configurations and manipulated three different configuration spaces for two different test suites. Our results suggest that there are only small differences, at the test suite level, in fault finding effectiveness across configurations. In addition, differences in block level code coverage are also small. Specification-based test suites had higher overall code coverage and showed greater variance in the metrics. At the test case level, however, these differences were magnified, suggesting that an entire test suite is more robust to changing configurations.

A small number of the faults detected in this study were configuration dependent. The implications for combinatorial interaction testing are that for a small subset of faults this type of testing may increase overall fault detection. In addition, for testing techniques such as regression testing in which only subsets of test suites may be used, the impact of changing configurations are greater, and interaction testing methods may improve overall fault detection effectiveness.

In future work we intend to extend the test suites utilized in this study to improve overall code coverage, and examine additional configuration spaces. We plan to compare metrics obtained from each of the configuration spaces with minimal subsets of configurations defined by combinatorial interaction testing. Finally, we plan to conduct additional studies on other user configurable systems.

Acknowledgments

This work was supported in part by an NSF EPSCoR FIRST award. We thank Zhimin Wang for his work on MyIE.

7. REFERENCES

[1] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. *AT& T Technical Journal*, 71(3):41–47, 1992.

[2] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*, 1998.

[3] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Supercomm/IC, IEEE International Conference on Communications*, pages 745 – 752, 1994.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[5] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 174–179, 1998.

[7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the International Conference on Software Engineering*, pages 285–294, 1999.

[8] S. R. Dalal, A. Jain, G. Patton, M. Rathi, and P. Seymour. AETG^{SM}web: a Web based service for automatic efficient test generation from functional requirements. In *Proceedings of the IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 84–85, Oct 1998.

[9] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proceedings of the International Conference on Software Engineering, (ICSE)*, pages 205–215, 1997.

[10] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *International Conference on Software Maintenance*, pages 169–179, 2001.

[11] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows System Symposium*, pages 1–10, July 2000.

[12] M. Grindal, J. Offutt, and S. T. Andler. Combination testing strategies: A survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[13] M. Hardojo, S. Elbaum, and Z. Wang. The effect of field data integrity on failure reproduction and fault isolation. In *Workshop on Remote Analysis and Monitoring of Software Systems*, pages 41–44, May 2004.

[14] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the NASA/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[15] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[16] R. Mandl. Orthogonal Latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, 1985.

[17] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the International Conference on Software Engineering*, pages 459 – 468, May 2004.

[18] B. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of MacOS applications using random testing. In *First International Workshop on Random Testing*, pages 46–54, July 2006.

[19] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31:678–686, 1988.

[20] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[21] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[22] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *International Symposium on Software Testing and Analysis*, pages 45–54, July 2004.

[23] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

[24] C. You. MyIE 3.1. http://www.myie2.com/html_en/home.htm.