

Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization

Xiao Qu, Myra B. Cohen, Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
{xqu,myra,grother}@cse.unl.edu

ABSTRACT

Configurable software lets users customize applications in many ways, and is becoming increasingly prevalent. Researchers have been creating techniques for testing configurable software, but to date, only a little research has addressed the problems of regression testing configurable systems as they evolve. Whereas problems such as selective retesting and test prioritization at the test case level have been extensively researched, these problems have rarely been considered at the configuration level. In this paper we address the problem of providing configuration-aware regression testing for evolving software systems. We use combinatorial interaction testing techniques to model and generate configuration samples for use in regression testing. We conduct an empirical study on a non-trivial evolving software system to measure the impact of changes in configurations on testing effectiveness, and to compare the effectiveness of different configuration prioritization techniques on early fault detection. Our results show that configurations can have a large impact on fault detection and that prioritization of configurations can be effective.

1. INTRODUCTION

User configurable software — software that can be customized through a set of options by the user — is becoming increasingly prevalent. A single user configurable software application can often be instantiated in an enormous number of ways. From a testing perspective, each configuration may appear largely similar, but the underlying execution of code for the same set of test cases may differ widely across configurations [10]. This increases the burden on software engineers, who must consider not just which inputs to utilize in testing, but also which configurations.

The impact of configurability can be particularly large in the context of regression testing, which is performed each time a system is modified and is often resource limited [2, 3, 19, 22]. To date, most regression testing research has treated software systems as if they possessed single homogeneous

configurations. A primary focus has been on techniques for reducing test suite size (regression test selection) (e.g., [6, 23, 26, 27]) or on ordering test cases (test case prioritization) (e.g., [13, 29, 32]). None of this research, however, has explicitly considered issues involving configurations.

We term the collection of all possible instantiations that comprise the configuration space for a software system the *configuration definition layer* (CDL) for that system. The CDL sits on top of the normal set of inputs to the system and therefore magnifies by a multiplicative factor the already large set of inputs needed for testing. Arguably, all possible settings of the CDL should be tested with each applicable input, but in practice this is infeasible. Alternatively, a sampling technique could be used to somehow “cover” the configuration space, and each input could be utilized on each of the selected configurations.

Recent work suggests that combinatorial interaction testing (CIT) may provide an effective way to sample configurations for testing [15, 18, 33]. Using CIT-based testing approaches, faults occurring under the sampled configurations can be revealed. Most of this prior work, however, has been concerned with the testing of single versions of software systems, rather than the effects that occur when regression testing multiple consecutive versions as a system evolves.

In this paper we address this lack, investigating approaches for configuration-aware regression testing. We quantify the impact of configurability on the effects of regression testing of software, measure the effectiveness of CIT sampling compared to random sampling in regression testing, and examine whether we can improve early fault detection during regression testing through configuration prioritization. To do this we study several versions of the evolving open source text editor `Vim`. Our results show that the CDL is important, and the choice of configurations for testing can impact the resulting fault finding ability of the regression test suite by as much as 70%. We also show that CIT test suites perform better than random ones based on the same configuration model, and that there is justification for considering configuration prioritization during testing.

The remainder of this paper is organized as follows. The next section provides motivation for and background on CIT and prioritization. Section 3 describes the specific prioritization techniques utilized in our empirical studies. Section 4 presents the study design, and Section 5 presents their results. Section 6 provides further analysis and discussion. Section 7 describes related work and Section 8 concludes.

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.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

2. BACKGROUND

In this section we provide background information on combinatorial interaction testing and test case prioritization.

2.1 Combinatorial Interaction Testing

CIT sampling models the inputs or configuration options for a software system (*factors*) and their associated *values* and combines these systematically so that all t -way ($t > 1$) combinations of inputs or options are tested together [7]. Here, t is called the strength of testing, and when $t=2$, we call this pair-wise testing.

CIT samples are defined by mathematical objects called covering arrays. A *covering array*, $CA(N; t, k, v)$, is an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t at least once [8]. Quite often in software testing the number of values for each factor is not the same. Therefore, we use the following expanded definition (often called a mixed level covering array) that uses a vector of vs for the factors.

A covering array, $CA(N; t, k, (v_1 v_2 \dots v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^k v_i$, where each column i ($1 \leq i \leq k$) contains only elements from a set S_i of size v_i and the rows of each $N \times t$ sub-array cover all t -tuples of values from the t columns at least once. We use a shorthand notation to describe these arrays with superscripts to indicate the number of factors with a particular number of values. For example, a covering array with five factors, three of which are binary and two of which have four values, can be written as follows: $CA(N; 2, 3^2 2^4)$. (We remove the k since it is implicit). Covering arrays have been shown to be effective test suites [4, 7, 17, 33].

To model software using CIT sampling we first need to describe the relevant factors and their associated values. One way to do this is with the Test Specification Language (TSL) [24], a specification based method for defining the combinations of factors influencing program behavior that should be tested together. TSL partitions the system inputs into *parameters* and *environment factors* and within these into *categories*. For each category, a set of *choices* is defined based on equivalence classes.

Though TSL was created to define combinations of program parameters and environment factors, it is not limited to this. TSL can also be used to define factors for covering arrays from system configurations, each of which is considered a category. Each of the choices in the categories becomes a value in CIT terminology.

In TSL the large combinatorial space formed by considering all combinations of categories and choices is reduced through two approaches. The first approach sets specific choices as *single* or *error*, meaning that these are tested alone. The second approach adds properties to particular choices and defines *constraints* that relate other choices to these properties. These approaches can significantly reduce the final set of combinations associated with a TSL specification. In TSL all possible combinations are then generated given the set of specified constraints. CIT techniques do not directly use these methods, but rather reduce the combinatorial space by systematically testing only t -way combinations. It is possible, however, to add single test cases to the CIT test suite and to consider constraints if certain combinations are illegal [7, 9].

Figure 1 shows an example of a partial TSL definition (without any single or error notations, or constraints) for

the text editor Vim, and an applicable pair-wise test suite. The shaded boxes are all 2-way combinations with the value “set et” for factor `expandtab`.

| Partial TSL for vim | | Pair-wise CIT Sample | | | |
|--|-----------|----------------------|-----------------|----------------|------------------|
| Properties expandtab: set et set noet smarttab: set sta set nosta tabstop: 8 1 textwidth: 0 2 78 | | expandtab | smarttab | tabstop | textwidth |
| set et | set sta | 8 | 78 | | |
| set et | set nosta | 1 | 78 | | |
| set et | set nosta | 1 | 2 | | |
| set noet | set sta | 1 | 0 | | |
| set noet | set sta | 8 | 2 | | |
| set et | set nosta | 8 | 0 | | |

Figure 1: A CIT Test Suite Defined using TSL

2.2 Test Case Prioritization

Commonly, test case prioritization is used in regression testing, at the test suite level, with the goal of detecting faults as early as possible in the regression testing process, given a test suite inherited from previous versions of the system. There are many techniques (e.g., [14, 20, 28, 30]) for prioritizing test cases based on various forms of information such as statement coverage, function coverage and fault finding probability.

Just as test cases can be prioritized, so can configurations, the motivation in this case being to order the configurations to be tested in a manner that helps meet testing objectives (e.g., fault detection) earlier. To our knowledge this idea has not yet been systematically explored. Bryce and Colbourn [5] present an algorithm to prioritize CIT test suites, but theirs is a general algorithm that may or may not apply to configurable software. Moreover, their approach does not address a key aspect of prioritization, namely, how to weight the various elements that drive prioritization on real software systems.

In [25], we examined prioritization of CIT test suites and developed several ways to control the prioritization through weightings. We used methods that leverage code coverage from prior releases, as well as one that is specification based. Additionally, we observed that Bryce and Colbourn’s prioritization technique is a combined generation and prioritization technique, rather than pure prioritization, because it does not simply re-order existing tests, but rather, regenerates them each time. We modified the algorithm to perform pure prioritization and compared the two approaches. Our results show that all prioritized test suites detected faults earlier than unordered ones. Since our configurations were generated by CIT techniques, these techniques can also be applied to prioritize configurations.

3. PRIORITIZING CONFIGURATIONS

Prior work on test case prioritization has focused primarily on ordering test cases. Our work is fundamentally different from this in that we are prioritizing configurations over a fixed set of test cases; we leave prioritization of the test cases as an additional and orthogonal challenge. Since our configuration model is based on combinations of configuration options, we need techniques that leverage information about the importance of interactions between options. In

[25] we prioritized CIT test suites for regression testing. We can modify that strategy to work on configurations.

We use Bryce and Colbourn’s algorithm [5] to generate prioritized test suites. The CIT test suites generated are a special kind of covering array called a *biased covering array*. The algorithm uses the *interaction benefit* or *importance* of individual factors and values to determine the final configuration order. The algorithm (see [5] for details) begins by computing a total interaction benefit for each factor. The factors are sorted in decreasing order of interaction benefit and then filled as follows. First, the individual interaction benefit for each of the factor’s values is computed. This selects the value of the factor that has the greatest interaction benefit. After all factors have been fixed, a single test has been created, and the benefits for factors are recomputed and the process starts again. The algorithm is complete when all pairs have been covered.

This type of prioritization is really a regeneration technique; it generates a new CIT sample for each new version. In [25] we extended the concept of interaction benefit to apply to previously generated test suites as a basis of prioritization. In this paper, we refer to the first technique as *regeneration* and the second as *pure prioritization*. Figure 2 shows the high level approach for our techniques. First we calculate *weights of importance* from either the previous version (when using code or fault based prioritization) or current version, and then we use that to either regenerate or prioritize the current set of configurations.

One requirement for using interaction benefit to drive prioritization is that we need to assign weights of importance to each factor and value in the CIT model. In [25] we used block coverage from the prior version to weight the values of the model. In this technique, once we decided which test cases contributed the most to finding faults we calculated the occurrence of individual values in those test cases and gave them the highest weights. We take a similar approach at the configuration level.

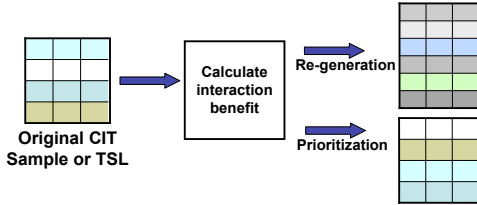


Figure 2: Prioritization/Regeneration Process

In a configurable system our model is at a higher level of abstraction than the test suite/test case level, so we first need some metrics that can help us assess how important an individual configuration is. We describe these next.

3.1 Metrics for Fault and Block Coverage

In [10] we developed a set of metrics for measuring the changes in a test suite’s fault detection behavior across different configurations. These metrics were derived from metrics developed by Elbaum et al. for code coverage [12]. We begin with a *block matrix* B and a *fault matrix* F . 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 (executed the code in) block i , otherwise the cell contains 0.

Let F be an $f \times t$ matrix, where f is the number of unique faults in the program and t is the number of test cases in the test suite. If cell (i, j) in F contains a 1, this means that test case j detected fault i .

Figure 3 shows examples of a block matrix and two fault matrices, F_1 and F_2 . In this example, there are four faults, three blocks and five test cases.

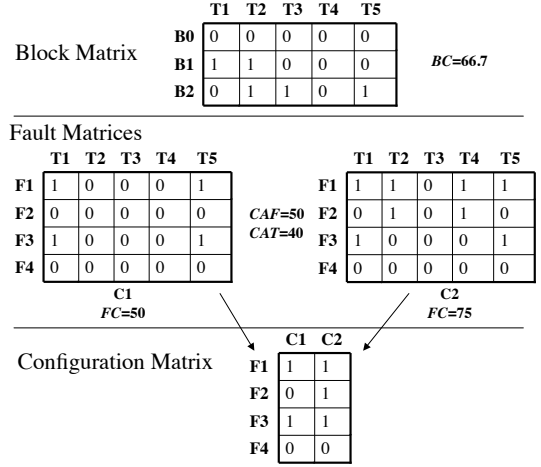


Figure 3: Block and Fault Matrices

We use these block and fault matrices to define and calculate two metrics: block coverage and fault coverage. We include two additional related metrics explained here and used later in our experimentation, coverage across faults and coverage across tests.

Block Coverage (BC)

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 cell in i contains a 1 (i.e. at least one test covers this block) and 0 otherwise. Then:

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

In Figure 3, the BC for matrix $B_1=66.7\%$.

Fault Coverage (FC)

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

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

Matrices F_1 and F_2 of Figure 3 have FC values of 50% and 75%, respectively.

Coverage Across Faults (CAF)

CAF compares a pair of fault matrices, F_1 and F_2 , and measures the percentage of rows that differ between them. Intuitively it tells us how much variance there is within the test suite for detection of faults by individual test cases between configurations. If rows $F_1 i$ and $F_2 i$ have at least one cell (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 3 is 50%.

In our study, CAF is calculated for all $\binom{n}{2}$ pairs of configurations where n is the number of configurations in our sample.

Coverage Across Tests (CAT)

CAT also compares two fault matrices, F_1 and F_2 , and measures the percentage of columns that differ between them. This metric tells us how much individual test cases vary between configurations with respect to fault detection. If columns F_1i and F_2i have at least one cell (j, i) that differs, $diff_Fcount_i = 1$. Then:

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

The CAT value for matrices F_1 and F_2 in Figure 3 is 40%. In our study we examine all $\binom{n}{2}$ pairs of configurations when using this metric.

3.2 Computing the Interaction Benefit

Given block and fault matrices as just described, we use a weighting scheme that relates these metrics to our configuration model. Initially we have one block and one fault matrix for *each* configuration. We first aggregate these into *configuration matrices*. Figure 3 shows how we combine two matrices into a single configuration matrix (we show this for two fault matrices but the same can be done with block matrices). The rows of the matrix are still faults (or blocks), but the columns are now configurations. A 1 in a cell means that at least one test case for that configuration found the fault (or covered the block for a block matrix). We use this matrix to determine the FC or BC for each configuration, and compare configurations, by summing the columns. In this example C2 has higher fault coverage than C1.

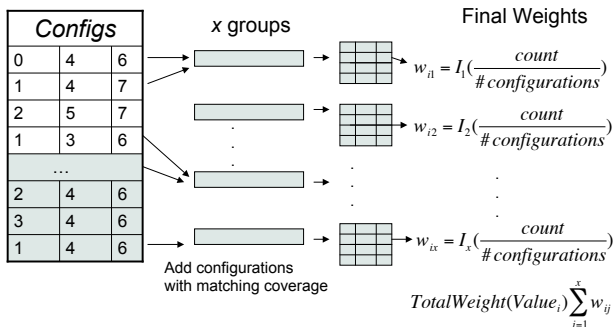


Figure 4: Setting the Weights for Interaction Benefit

To prioritize configurations, we use a greedy approach. We order our configuration matrix in descending order of FC or BC values and gather the first x configurations that provide the maximum coverage for this matrix (see Figure 4), re-ordering between each selection. We place all equivalently important configurations in sets (rather than randomly selecting one); i.e. for each of the x configurations we find all other configurations that provide equivalent coverage. This is shown as the second stage in Figure 4. For instance if the first configuration has an FC value of 45%, we find all other configurations that also have this FC value. We then examine the next configuration. If this adds $k\%$ more FC coverage we find all other configurations with the same additional coverage, and so on. We select configurations without replacement (each can belong to only a single group).

We next calculate weights for each value. We sum the number of times a value occurs in each of the groups and

divide this by the number of configurations in the group. For instance if value 1 occurs two times and we have four configurations in this group it has a weight of .50, and if value 2 occurs twice it has a weight of 1.0.

For the final stage, we first assign an importance, I , to each of the x groups. The importance is set to the cumulative fault coverage of that group divided by the total fault coverage (i.e. it is just the percentage of fault coverage it provides in relation to the maximum for the CIT sample). We use a similar approach for block coverage. As shown in the rightmost part of Figure 4, the weight of each value is multiplied by its importance I , and the x weights are summed to obtain the individual weight for that value.

The final weights are then fed into the regeneration algorithm which computes the interaction benefit [5].

3.3 Specification Based Weightings

There may be instances where we do not have prior code coverage, but want to prioritize based on the current system specifications. To account for such a situation in [25] we used a specification based approach for prioritizing CIT test cases for regression testing. Since we define our configuration model using TSL we can use this for weighting as well. In TSL based prioritization we have the advantage of not requiring a prior version; instead we can rely on our existing version of the software to produce information to drive the prioritization.

Our technique for setting weights works as follows. For each category in the TSL specification we examine that category's possible choices. In the case of binary choices, where one choice turns a feature on and one turns a feature off, we set the ON option to a weight of 0.9 and the OFF option to 0.1. Our intuition is that the ON option will cause more code to be executed.

In cases where we have multiple choices for a category, we use a greater number of features or higher complexity of the choice as a proxy for higher code coverage. For instance in the subject `Vim` that we use in our studies described later, we have a category called `laststatus` which determines when the last window will contain a status line. There are three options: `never`, `only when 2 windows` and `always`. We assign the highest importance (0.5) to the last option, a medium importance to the second (0.3), and the smallest importance (0.1) to the first. The absolute values used in the prioritization algorithm are less important than the relative values, therefore we have chosen these to reflect a relative importance and have not fine tuned them to specific values. We realize that different heuristics and different absolute values may impact the quality of this method. We leave investigation of this for future work.

3.4 Pure Prioritization

For this method, we use a greedy approach to order the configurations. We can use any of the weighting schemes above. We repeatedly calculate the interaction benefit for each configuration (this differs from [5] because their regeneration algorithm calculates benefit only for the factors and values). We calculate this by summing the product of each pair of weights for the configuration. We select (using randomness to break ties) the configuration that has the highest interaction benefit. We continue this process until we have ordered all of the configurations.

4. EMPIRICAL STUDIES

We have designed a set of empirical studies to investigate the impact that the CDL has on regression testing, to measure the applicability of using CIT as a sampling mechanism, and to assess the effectiveness of prioritization. Our studies are designed to answer the following research questions:

RQ1: What is the effect of changing configurations on the outcome of regression testing across consecutive versions of a program?

RQ2: Is CIT an effective method for sampling configurations for testing?

RQ3: Can we improve fault detection effectiveness, when resources are constrained, through prioritization of configurations?

The rest of this section describes our objects of analysis, our independent and dependent variables, our methodology, and threats to validity. Subsequent sections present and analyze results.

4.1 Objects of Analysis

As an object of analysis we selected `Vim`, an open source, multi-platform text editor extended from `vi` written in C [21]. The object was obtained as a pre-release from the Software Infrastructure Repository (SIR) [11] and is augmented with a test suite containing 975 test cases, organized into groups by functionality, and hand seeded faults. We selected this object because it is highly configurable, has a non-trivial code base and has both a default configuration and a test suite that was not developed for this study. This helps to reduce the potential sources of bias for our results.

We conduct our experiments on a set of consecutive versions of `Vim`, which represents six releases of the software (version 2-7 from SIR). This corresponds to `Vim` versions 5.3-5.8, developed during a three year window from 1998-2001.

Table 1 provides basic information on our object of analysis, including the number of basic blocks in system versions as calculated by `gcov`[16], the number of source lines of code calculated by `sloccount`[31], and the number of methods changed or added between versions. The number of faults in each version is also shown. Since relatively few faults (2-4) were seeded in any single version, we added 10 additional faults in each, through the use of a mutation testing tool [1]. To avoid bias, we generated all possible mutations for each version and then randomly selected (with replacement) 10 modules for mutation, in which at least one method had been changed between versions. Within each of these 10 modules a mutation from within the changed methods was randomly selected and added to our object.

| Version | # blocks | loc | # changed methods | Faults hand+mut |
|---------|----------|---------|-------------------|-----------------|
| v2 | 26,081 | 87,442 | 664 | 3+10 |
| v3 | 30,217 | 105,225 | 830 | 3+10 |
| v4 | 30,426 | 105,690 | 323 | 3+10 |
| v5 | 30,553 | 106,391 | 317 | 4+10 |
| v6 | 30,744 | 107,992 | 324 | 2+10 |
| v7 | 30,764 | 105,944 | 311 | 3+10 |

Table 1: Experimental Object

`Vim` has a user configuration file `.vimrc` that controls a set of from 146 to 187 user configurable options for the different versions of the software. We used the online documentation found at [21] along with the `-setall` option within the

software to list and model the system’s configuration space, using TSL to specify the configuration options. The original `Vim` test suite from SIR was designed to be run with a single default configuration. In each version there were some configuration options that either did not allow certain test cases in the original suite to run (for instance an option that turns on interactive mode will not work because our test cases run in batch mode), or that we deemed to have little effect on the software under test such as modifying a path location for a specific directory; we ignored these options. In total we modeled 90 options in version 2, 93 options for version 3, 4, 6, 7 and 76 options for version 5. Note that in version 5 there was a significant decrease in the configurable option space which reduced our model for this version.¹

Next we used the TSL definition (following the process reported in [25]) to generate a CIT configuration sample for each version. Due to the size of the configuration space and time required for testing we used a strength 2 (or pair-wise) CIT sample. We used a $CA(60; 2, 2^{77}3^74^26^310^1)$ array for version 2, a $CA(60; 2, 2^{80}3^74^26^310^1)$ array for version 3, 4, 6, and version 7, and a $CA(60; 2, 2^{65}3^64^16^310^1)$ array for version 5. Note that each sample created has 60 configurations. We used a simulated annealing algorithm [8] to generate the samples. The configurations were then mapped to the `.vimrc` file for manipulation during experimentation. In addition to the covering array samples we generated a comparison sample of the same size for each model consisting of 60 random configurations.

4.2 Independent Variables

Our independent variables for RQ1 and part of RQ2 are the individual configurations in our covering array samples or random samples generated without replacement from all possible configurations of the TSL model. We refer to the original unprioritized covering arrays as `ca` and refer to the sample of 60 random configurations as `rand`.

To answer part of RQ2, and for RQ3, we use the prioritization techniques described in Section 3 as well. We refer to these techniques in terms of both the weighting technique and whether or not the array was prioritized or regenerated: **pure-BC** and **regen-BC** for block coverage weighting, **pure-FC** and **regen-FC** for fault-coverage weighting. The last two techniques, **pure-TSL** and **regen-TSL**, reflect the weighting scheme that uses only the TSL to determine interaction benefit.

4.3 Dependent Variables

To address RQ1 and RQ2 we measure block coverage (**BC**), fault coverage (**FC**), coverage across faults (**CAF**) and coverage across tests (**CAT**) as described in Section 3.

For RQ2 we also compare the minimal number of configurations in an unordered sample needed to detect the same number of faults detected by all configurations in that sample (**NCF**).

Finally, to address RQ3, we examine the *Normalized Percentage of Faults Detected* (**NAPFD**) which was first described in [25]. In prior work [14, 28], a metric that has been

¹On further analysis, it appears that version 5 was set up by default to use a minimal subset of options, but that the other options are still available within the software. We chose to ignore this in order to maintain the integrity of our model developed through documentation, rather than by examination of source code.

commonly used for prioritization is the *Average Percentage of Faults Detected* or APFD. This metric measures the area under the curve when the percentage of faults found is plotted on the y -axis against the percentage of the test cases run on the x -axis. NAPFD normalizes this metric to work when we have differing numbers of test cases or faults. The NAPFD formula is as follows:

$$NAPFD = p - \frac{TF_1 + TF_2 + \dots + TF_m}{m \times n} + \frac{p}{2n}$$

To illustrate, Table 2 shows a fault matrix with five test cases and eight faults. The ordering of the test cases is T_3, T_5, T_2, T_4, T_1 . The first test case (or 20% of the test cases) finds three faults. After three test cases or 60% of the test cases have been executed, five faults have been found (62.5%).

In the NAPFD formula n represents the number of test cases, m represents the total number of faults found by the test suite. We use a proportion, p , to represent the number of faults detected by the test suite within our budget divided by the total number of faults detected in the full test suite. TF_i stands for the index of the test case (when testing in prioritization order) in which Fault i was found. For instance, using the above order, TF_2, TF_4 and TF_7 all have a value of 1, while TF_5 and TF_6 are 2. If a fault i is never detected, we set $TF_i = 0$.

In our example, if we are able to run the whole test suite, $m = 8, n = 5, p = 1, TF_1 = TF_3 = 4, TF_2 = TF_4 = TF_7 = 1, TF_5 = TF_6 = 2$ and $TF_8 = 5$, the NAPFD value is 0.6. But if we only have a budget to run 3 test cases, the n changes to 3 and TF_1, TF_3 and TF_8 change to 0. Now the NAPFD value is 0.44.

| | T_1 | T_2 | T_3 | T_4 | T_5 |
|-------|-------|-------|-------|-------|-------|
| F_1 | | | | x | |
| F_2 | | x | x | | |
| F_3 | | | | x | |
| F_4 | | | x | | x |
| F_5 | | | | | x |
| F_6 | | | | | x |
| F_7 | x | | x | | |
| F_8 | x | | | | |

Table 2: Test Order: T_3, T_5, T_2, T_4, T_1

4.4 Study Methodology

For our first set of experiments we run the entire test suite on each configuration without any faults, letting outputs serve as the oracle, and then we turn on each fault individually and measure fault detection, to alleviate the potential masking of one fault by another. For each model, we run each set of experiments under each configuration as is defined by the covering array or random array, as well as the default configuration (not created by us) from SIR. We collect block coverage on the fault free version using `gcov`.

For all of our prioritization experiments, that use weights based on block coverage and fault detection ability, we use program $P-1$ to prioritize configurations for program P , but for the TSL based weighting we use data on the current version, P , as the source for prioritization information. When collecting data on the unordered CIT or random samples for

NCF and for the unordered prioritization results, instead of using a single number we randomly select 50 orders for the sample and use the average of the results, to reduce sources of bias in the generation process.

4.5 Threats to Validity

Empirical studies are subject to threats to validity. We have attempted to reduce these through our experiment design, however, we outline the major threats here. With respect to external validity (or the difficulty of generalizing to other objects), we have examined only one software system, written in C, and results obtained with other systems may not match these. Similarly, we have utilized only one unconstrained TSL definition. With respect to internal validity (the possibility that factors other than variance in our independent variable is responsible for our results) our greatest concern is problems with our instrumentation, and thus we have manually cross-validated our analysis programs on small examples and manually validated random selections from the real results. With respect to construct validity (the validity of measures), there may be other metrics that are more accurate in measuring the cost-benefits of regression testing techniques.

5. RESULTS

We now analyze the results of our study with respect to our three research questions.

5.1 RQ1: The Effect of Configurations on Regression Testing

To address RQ1 we examine the fault detection ability and code coverage of test suites under different configurations across the sequence of Vim versions, by measuring the FC and BC data values for each configuration. Figures 5 and 6 provide box plots for FC and BC across all versions of Vim. Additionally, Figure 7 provides similar data but shows the actual number of faults found in each sample. Table 3 shows the detailed FC data. In Figure 5 the dot on each box plot is the FC value for the default configuration provided from SIR. In most versions the default configuration has a level of performance well below that of the best configuration. For all versions, the FC values range from about 30% to 100%, representing a range of almost 10 faults, although the BC values are fairly constant.

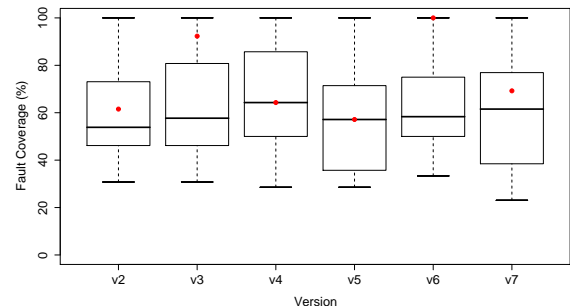


Figure 5: Fault Coverage

The CAF (Figure 8 and Table 4) and CAT (Figure 9 and Table 5) results are used to distinguish behaviors relative to individual faults and tests, focusing on a different granularity of our measures. The data shows a great deal of

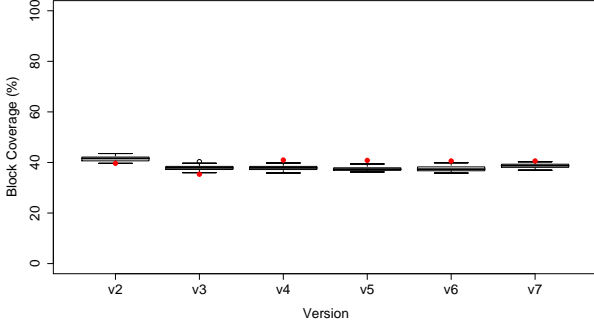


Figure 6: Block Coverage

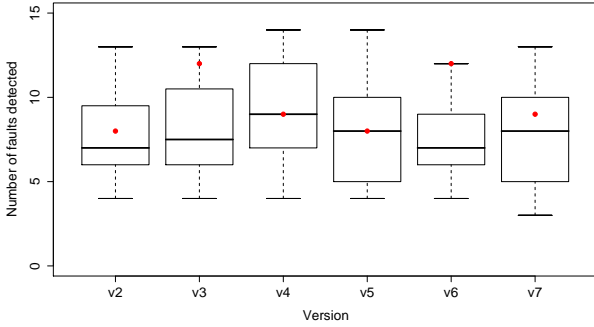


Figure 7: Fault Coverage by Number

| | Min | Max | Mean | Median | Mode | StdDev |
|----|-------|--------|-------|--------|-------|--------|
| v2 | 30.77 | 100.00 | 58.46 | 53.85 | 46.15 | 17.76 |
| v3 | 30.77 | 100.00 | 61.15 | 57.69 | 50.00 | 21.22 |
| v4 | 28.57 | 100.00 | 65.48 | 64.29 | 64.29 | 20.72 |
| v5 | 28.57 | 100.00 | 58.81 | 57.14 | 57.14 | 21.70 |
| v6 | 33.33 | 100.00 | 63.75 | 58.33 | 50.00 | 21.47 |
| v7 | 23.08 | 100.00 | 60.00 | 61.54 | 46.15 | 21.84 |

Table 3: Statistics for FC

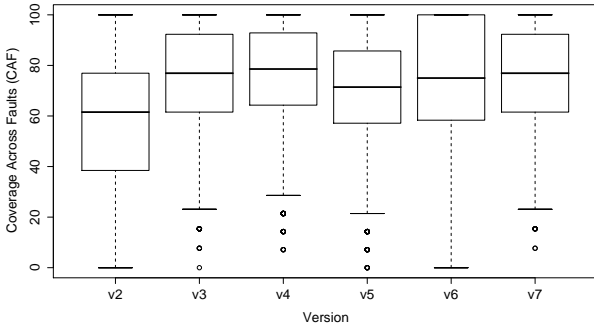


Figure 8: Coverage Across Faults

variation in our CAF values. This means that some configurations match closely in fault detection effectiveness across versions while others vary greatly. The average CAF value for each version is between 60% and 75%. The CAT values exhibit a much smaller difference, averaging only around 3% except on version 4 which exhibits a large fluctuation.

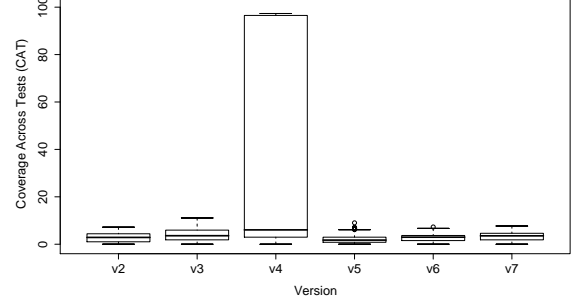


Figure 9: Coverage Across Tests

| | Min | Max | Mean | Median | StdDev |
|----|------|--------|-------|--------|--------|
| v2 | 0.00 | 100.00 | 59.55 | 61.54 | 24.23 |
| v3 | 0.00 | 100.00 | 74.02 | 76.92 | 21.33 |
| v4 | 7.14 | 100.00 | 75.23 | 78.57 | 22.29 |
| v5 | 0.00 | 100.00 | 69.21 | 71.43 | 23.41 |
| v6 | 0.00 | 100.00 | 73.96 | 75.00 | 24.06 |
| v7 | 7.69 | 100.00 | 74.66 | 76.92 | 22.18 |

Table 4: Statistics for CAF

| | Min | Max | Mean | Median | StdDev |
|----|------|-------|-------|--------|--------|
| v2 | 0.00 | 7.18 | 2.79 | 2.87 | 1.82 |
| v3 | 0.00 | 11.08 | 3.99 | 3.59 | 2.50 |
| v4 | 0.00 | 97.33 | 45.41 | 6.05 | 46.41 |
| v5 | 0.00 | 9.02 | 2.10 | 1.74 | 1.52 |
| v6 | 0.00 | 7.28 | 2.66 | 2.97 | 1.55 |
| v7 | 0.00 | 7.69 | 3.32 | 3.54 | 1.70 |

Table 5: Statistics for CAT

Our results bring us to the conclusion that the CDL plays an important role in fault detection on this system, and *which* configuration is tested greatly impacts the result of regression testing. There are large differences in fault detection at the test suite level, although there are only small differences in block coverage. We see the greatest fault detection variation at the test suite level of granularity.

5.2 RQ2: Effectiveness of CIT for Testing Configurations

To examine our second research question, we first compare fault detection abilities between two sets of configurations: our CIT sample and a randomly generated sample of the same size based on the same TSL model.

Both of these samples detect all seeded faults when all configurations are tested cumulatively. However, there are some differences in the distribution of fault finding effectiveness abilities for individual configurations. Figure 10 provides a box plot comparison of the two sets of configurations. We can see that the CIT sample seems to be distributed more towards the higher fault finding end.

We examined this further by finding the number of configurations needed to detect all faults within each sample if we were to just run each sample in an unprioritized order (NCF). The box plot in Figure 11 shows 50 randomly selected orders for each sample. We show this for all versions.

We also applied the Wilcoxon two sample test on each version, with an α level of 0.05, to examine whether there is significant difference between these two groups (see Table 6). Our data shows a significant difference.

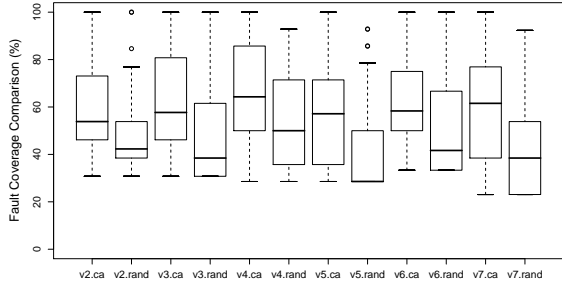


Figure 10: CIT vs Random

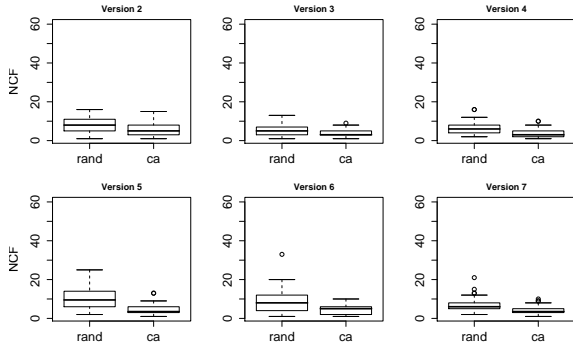


Figure 11: CIT vs Random NCF

We can come to only a weak conclusion on this research question. CIT appears to be effective, but additional studies are needed to validate this. Based on this set of experiments, CIT clearly has some benefit and slightly outperforms the random sample. The results of fault detection using CIT generated configurations show that in most cases, the median fault finding ability of the CIT sample is similar to or higher than that of the default SIR configuration with respect to the number of faults detected. Half of the CIT configurations detect more faults than the default SIR configuration. Though the cumulative fault finding effectiveness of the random sample is the same as that of the CIT sample, individually the CIT configurations appear to have slightly better fault detection capabilities.

5.3 RQ3: The Effectiveness of Prioritization

RQ3 examines whether prioritized CIT configurations yield faster fault detection than unprioritized ones.

We provide the NAPFD values for all of the prioritization techniques we considered in Table 7. To simulate a resource-constrained testing environment, we selected budgets of configurations in increments of five configurations for analysis. We calculated the NAPFD for each method at each budget level. After the 15th configuration, all faults are found in all versions, so the NAPFD stabilizes and is very close to the full budget of 60; thus, we do not show the data for increments in between. For all versions except version 5, the prioritized and regenerated configurations exhibit better NAPFD values than unordered CIT configurations, although the difference decreases as we run more samples in the configuration. We see very little difference between the different weighting schemes for pure prioritization. The regenerated samples, however, always seem to always provide

| Wilcoxon Two-Sample Test | | |
|--------------------------|---------|-----------|
| Statistic | | 1843.5000 |
| Normal Approximation | | |
| Z | | -4.7360 |
| One-Sided | Pr < Z | < .0001 |
| Two-Sided | Pr > Z | < .0001 |
| t Approximation | | |
| One-Sided | Pr < Z | < .0001 |
| Two-Sided | Pr > Z | < .0001 |

Table 6: Wilcoxon Test Results for v2

| F/T | T ₀ | T ₁ | T ₂ | T ₃ |
|---------------------|----------------|----------------|----------------|----------------|
| F ₀ | 0 | 0 | 0 | 0 |
| F ₁ | 1 | 0 | 0 | 0 |
| F ₂ | 1 | 1 | 0 | 0 |
| Fault Matrix for c1 | | | | |
| F/T | T ₀ | T ₁ | T ₂ | T ₃ |
| F ₀ | 1 | 1 | 1 | 1 |
| F ₁ | 1 | 0 | 0 | 0 |
| F ₂ | 1 | 1 | 0 | 0 |
| Fault Matrix for c2 | | | | |

Table 8: Fault Matrices for Configuration 1 and 2

the highest NAPFD values.

Our results suggest that both prioritized and regenerated configurations detect faults earlier than unordered configurations and that the regeneration techniques work better for early fault detection in our subjects. Which technique is used seems less important than whether we used pure prioritization or regeneration. The improvements in early fault detection by prioritization are not remarkable though; unordered configurations already exhibit a strong ability to detect faults early.

6. DISCUSSION AND FURTHER ANALYSIS

In this section, we provide additional discussion and analysis of the results just described. We also analyze some specific faults in detail to help further explain the results.

6.1 Configuration Dependent Faults

Inspecting our results and data further, it is apparent that some faults are found by every configuration while other faults are found by only some. To better understand this trend we plotted the faults for two versions of vim to show their distribution across configurations. Figure 12 shows the data for versions 4 and 6. On the y-axis we number the configurations and on the x-axis we list the individual faults. The size of the dots represents the number of test cases that detected a specific fault. The graphs show, for instance, that Faults 1, 2, 11 and 12 in version 4 are found during testing in all configurations. However, while Fault 12 seems very evenly handled across configurations, Fault 1 exhibits some variation. Other faults such as Fault 7 in version 4 are found with great frequency in some configurations but completely missed in others. We call the first type of fault a *configuration-independent fault* and the second type a *configuration-dependent fault*.

Note that there may be two definitions of configuration-independence, one at the test suite level and one at the test case level. It is possible for all configurations to find the same fault with at least one test case, but the actual test

| version | unordered | pure-FC | pure-BC | pure-TSL | regen-BC | regen-TSL |
|------------------------------|-----------|---------|---------|----------|----------|-----------|
| NAPFD values (budget: 5/60) | | | | | | |
| v3 | 76.05 | 77.69 | 77.69 | 80.77 | 90.00 | 90.00 |
| v4 | 76.36 | 90.00 | 90.00 | 90.00 | 90.00 | 90.00 |
| v5 | 73.88 | 71.43 | 68.57 | 67.86 | 90.00 | 90.00 |
| v6 | 72.98 | 90.00 | 90.00 | 90.00 | 90.00 | 90.00 |
| v7 | 76.15 | 90.00 | 73.08 | 90.00 | 90.00 | 90.00 |
| NAPFD values (budget: 10/60) | | | | | | |
| v3 | 87.58 | 88.85 | 88.85 | 90.38 | 95.00 | 95.00 |
| v4 | 87.81 | 95.00 | 95.00 | 95.00 | 95.00 | 95.00 |
| v5 | 86.41 | 80.71 | 79.28 | 78.57 | 95.00 | 95.00 |
| v6 | 85.83 | 95.00 | 95.00 | 95.00 | 95.00 | 95.00 |
| v7 | 87.78 | 95.00 | 86.54 | 95.00 | 95.00 | 95.00 |
| NAPFD values (budget: 15/60) | | | | | | |
| v3 | 91.72 | 92.56 | 92.56 | 93.59 | 96.67 | 96.67 |
| v4 | 91.88 | 96.67 | 96.67 | 96.67 | 96.67 | 96.67 |
| v5 | 90.88 | 87.14 | 86.19 | 85.71 | 96.67 | 96.67 |
| v6 | 90.56 | 96.67 | 96.67 | 96.67 | 96.67 | 96.67 |
| v7 | 91.86 | 96.67 | 91.03 | 96.67 | 96.67 | 96.67 |
| NAPFD values (budget: 60/60) | | | | | | |
| v3 | 97.93 | 98.14 | 98.14 | 98.53 | 99.17 | 99.17 |
| v4 | 97.97 | 99.17 | 99.17 | 99.17 | 99.17 | 99.17 |
| v5 | 97.72 | 96.79 | 96.55 | 96.90 | 99.17 | 99.17 |
| v6 | 97.64 | 99.17 | 99.17 | 99.17 | 99.17 | 99.17 |
| v7 | 97.96 | 99.17 | 97.76 | 99.17 | 99.17 | 99.17 |

Table 7: NAPFD values

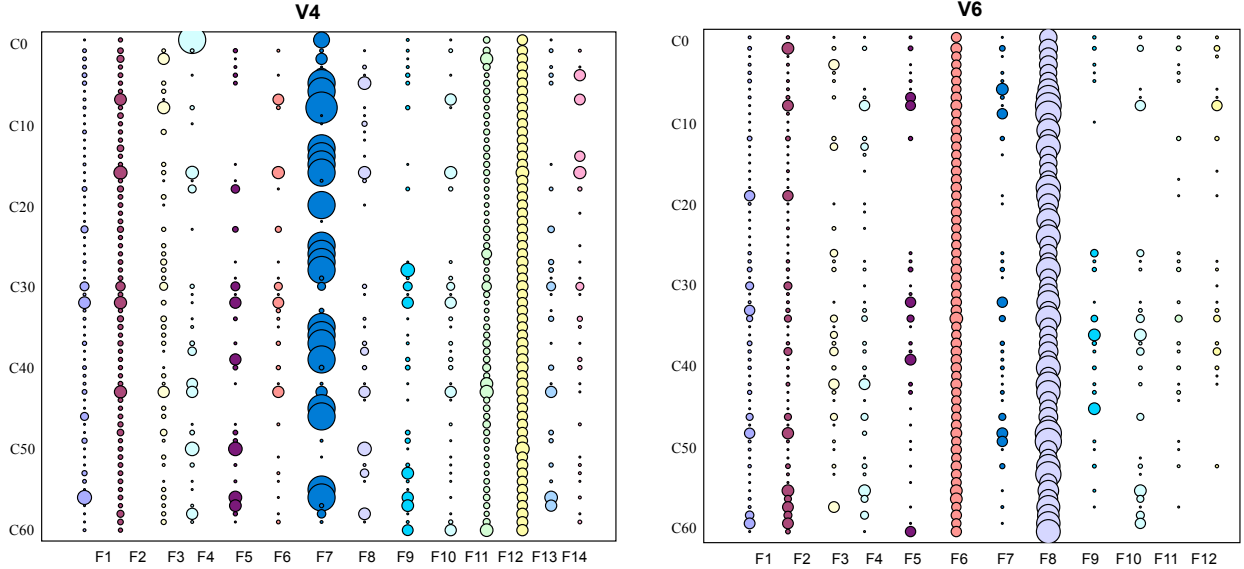


Figure 12: Fault Density

cases which find these may differ. This has implications for test case selection, since it implies that different test subsets may behave differently even when full test suites do not.

All of the configuration-independent faults found in our subject are of the first type; they are test suite configuration-independent. Although they are found under all configurations, they all vary by configuration as to which test cases detect them.

We next examine some specific faults to understand what makes them independent or dependent. Fault 11 in version 4, located in a routine named `window.c`, is a configuration independent fault. The faulty code incorrectly tests to see if a buffer is NULL. The code is called unconditionally, and its execution status has nothing to do with any options that

can be tuned by users. The test case that detects this fault under each configuration belongs to those that test program startup behavior, and thus it is run in each configuration.

Fault 7 in version 4 is configuration dependent; in fact, this fault plays a large role in the unusual fluctuation of the CAT metric for this version. This fault is located in a function named `my_sync()` in file `memfile.c`. This function is used to synchronize output of changed parts of the memory file to disk. Fault 7 is a particularly easy fault to find when its enclosing function is executed because it does not occur inside of any conditional statements. Thus, we analyzed the functions that call `mf_sync()` and found that they are all called under certain conditions, controlled by the parameters `p_uc` or `b_may_swap` which relate back to our TSL

model. The first parameter is associated with the number of synchronized characters, which is set by the configurable option `updatecount`, and the latter decides whether or not a swap file can be opened, which is turned on/off by another user configurable option called `swapfile`. In other words, if a swap file is not permitted by `set updatecount=0` and `set noswf`, then `mf_sync()` will never be executed and the fault will not be detected.

Now consider the alternative case where `updatecount>0`. The function `ml_open_file()`, which also calls `mf_sync()`, is called by another function, `set_bool_option()`. The `set_bool_option()` function is executed in any program run, because it is used to set Boolean options on startup. Hence, if we ever execute `set_bool_option()`, we will always execute `mf_sync()`, the faulty code, and detect the fault. Therefore, it is not surprising that a large proportion of test cases detect this fault when the required configurable options are set.

6.2 Analysis of Data Outliers

There are two places in our data that warrant further examination. First, in version 4 we see an unusual variation in the CAT metric. Second, we see an inverse in the results of NAPFD in version 5 for smaller budgets in the purely prioritized samples. We examine each of these in turn.

In version 4, there is a moderate range of CAF values but a very large range of CAT values. In fact this metric varies from almost 100 down to 6 (see Table 5). We illustrate how this is possible. Suppose we have four test cases, three faults and two configurations as shown in the fault matrices in Table 8. In this case the value of CAF is $1/3$ and the value of CAT is 1. The difference is caused by Fault 0. None of the test cases detect this fault under configuration 1 but all four test cases detect it under configuration 2. Hence this suggests that if there is a small range of CAF values but a large range of CAT values, there may exist faults that are triggered primarily by configuration options. This explains the phenomenon seen in version 4: Fault 7 is detected by more than 70% of the test cases under some configurations but never detected under others.

Finally we examine the anomaly in prioritization for version 5. As can be seen in Table 7, this version has lower NAPFD values than the unordered samples; but we see this only for the purely prioritized sample, not for the regenerated one. Between version 4 and version 5 a large number of options were deleted from the model (this was based on documentation of the tool). Since we use version 4 to prioritize version 5 for the pure prioritization, a lot of information is lost. During the calculation of the interaction benefit we may have a lot of options that do not contribute to the order of version 5. In the regenerated version we use the documentation from the current version, therefore, our prioritization seems to work as well as on the other versions.

7. RELATED WORK

There has been a large body of work on regression testing for both test suite selection (e.g., [6, 23, 26, 27]) and test suite prioritization (e.g., [13, 29, 32]). We do not attempt to list all of it here. Most of this work has focused on the test suite or test case as the object of selection or prioritization. This work differs in that it focuses on prioritizing configurations rather than the test suite. Other work on CIT for

testing configurable systems [15, 18, 33] examines the effectiveness of CIT to model configurations for testing, but it looks at fault detection or fault localization only in single versions of a software system and does not try to quantify the impact on the CDL. In [10] we examine the impact of configurations on the CDL, but this is a preliminary study on a single subject for a single version of a web browser. This work does not use prioritization, nor does it leverage CIT. Recent work on prioritization of CIT test suites for evolving systems was performed in [25]. This is closely related in that we have used similar prioritization techniques. However, the object of prioritization in that work is the test suite.

Finally, there has been some work on prioritization algorithms [5] for CIT. An algorithm from that work is used in our own work; however [5] does not present any empirical results of using the algorithm, nor does it provide methods to assign weights to calculate interaction benefits.

8. CONCLUSIONS

In this paper we have presented the results of an empirical study to examine the effects of changing configurations on a user configurable system, `Vim`, across multiple versions. We see compelling evidence that the CDL plays an important role in fault detection on this software subject. As many as 10 faults might be missed if the “wrong” configurations are omitted from testing. We see the greatest fault detection variation at the test case level which may have implications for regression test selection. Not one of the faults in our study were detected by an equivalent subset of test cases across all configurations.

We have also examined the effectiveness of CIT in sampling the configuration space. We weakly conclude that it is effective, but this needs to be validated through a more complete study. Finally, we have evaluated several techniques for CIT configuration prioritization. Our results suggest that both prioritized and regenerated configurations may detect faults earlier than unordered configurations and that the regeneration based techniques outperform the pure prioritization ones.

In future work we intend to examine additional configurable systems and apply these techniques. We are examining additional heuristics to be used for prioritization and examining the difference between random and CIT samples more thoroughly. We are also considering techniques for prioritizing higher strength covering arrays, and grouping homogeneous options for generating covering arrays with variable strength. Finally, we are examining the differences between configuration dependent and independent faults more closely.

9. REFERENCES

- [1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.
- [2] B. Beizer. *Black-Box Testing*. John Wiley and Sons, New York, NY, 1995.
- [3] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Reading, MA, 2000.

- [4] 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.
- [5] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, 48(10):960–970, 2006.
- [6] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 211–220, May 1994.
- [7] 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.
- [8] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 38–48, May 2003.
- [9] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, July 2007.
- [10] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. In *Proceedings of the International Workshop on Advances in Model-based Testing (AMOST)*, pages 1–9, Nov. 2006.
- [11] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
- [12] 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.
- [13] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112, Aug. 2000.
- [14] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.
- [15] S. Fouché, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *The Supplemental Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 557–560, Sept. 2007.
- [16] Free Software Foundation. gcov. <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, 2007.
- [17] 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.
- [18] 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.
- [19] H. Leung and L. White. Insights into regression testing. In *International Conference on Software Maintenance*, pages 60–69, Oct. 1989.
- [20] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, Apr. 2007.
- [21] B. Moolenaar. Vim. <http://www.vim.org/>, 2007.
- [22] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Saganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.
- [23] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Nov. 2004.
- [24] 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.
- [25] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 255–264, Oct. 2007.
- [26] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, Oct. 2004.
- [27] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.
- [28] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.
- [29] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.
- [30] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2006.
- [31] D. Wheeler. SLOccount. <http://www.dwheeler.com/sloccount/>, 2007.
- [32] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *International Symposium on Software Reliability Engineering*, pages 230–238, Nov. 1997.
- [33] 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.