Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces

Cemal Yilmaz
University of Maryland
cyilmaz@cs.umd.edu

Myra Cohen
University of Auckland
myra@cs.auckland.ac.nz

Adam Porter
University of Maryland
aporter@cs.umd.edu

Abstract

We have developed the Skoll system and process for performing distributed, continuous quality assurance. The goal of this system and process is to leverage remote user resources around-the-world, around-the-clock to provide developers much greater insight into their systems correctness, performance, and usage patterns.

Skoll is focused on supporting systems with large configuration spaces, that change frequently, and that are constructed with limited resources. In these situations, the cost and complexity of QA explodes because, in a very practical sense, there isn't just one system, but a multitude of related systems. Thus, bugs may appear in certain configurations, but not in others.

In previous work we use Skoll to automatically characterize configurations in which failures manifest. We showed that this information helped developers quickly narrow down the cause of failures, improving turn-around time for fixes. Our approach, however, did not scale well because they require us to exhaustively test each configuration in the configuration space. Therefore, in this paper, we use a mathematical object called a covering array to obtain relatively small test schedules with certain coverage properties over the entire configuration space. We empirically assess the effect of using covering array-derived schedules on the resulting fault characterizations.

1. INTRODUCTION

Many modern software systems must be customized to specific run-time contexts and application requirements. To support such customization, these systems provide numerous user-configurable options. For example, some web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) can have dozens, even hundreds, of options. While this flexibility promotes customization, it creates many potential system configurations, each of which may need extensive QA to validate. We call this problem *software configuration space explosion*.

To address this issue we have developed Skoll [8]– a distributed continuous QA process supported by automated tools which leverages the extensive computing resources of worldwide user communities in order to efficiently, incrementally and opportunistically improve software quality and to provide greater insight into the behavior and performance of fielded systems. Skoll does this by dividing QA processes into multiple subtasks which are then intelligently distributed to client machines around the world, executed by them, and their results returned to central collection sites where they are fused together to complete the overall QA process.

One QA task implemented in Skoll was to determine which specific options and option settings caused specific failures to manifest. We call this *fault characterization*. Fault characterization is done by testing numerous different configurations and feeding the results to a classification tree analysis. The output is a model describing the options and settings that best predict failure. For example, using Skoll on a CORBA implementation we found that if and only if the executable is running on Linux, with Corba Messaging Support enabled, but with Asynchronous Messaging optimization disabled, then socket connections timeout.

We gave this information to the system's developers who then quickly pinpointed the failure's cause. Further analysis showed that this problem had in fact been observed previously by several users, but that the developers simply hadn't been able to track down the problem. Skoll's automatically derived fault characterization, however, greatly narrowed down the search space, making the developers' job much easier.

While we were pleased with this outcome, the fundamental downside of this approach was that we have to test the entire configuration space. In the example cited above, for instance, that means that nearly 19,000 times, remote clients downloaded, configured and compiled the 1M+ lines of code system, and then executed a battery of tests. For each client this took about 6–8 hours. Furthermore, this was only a small subset of the system's entire configuration space. The actual space is much bigger. Clearly, some more efficient process will be necessary in general.

This paper proposes and evaluates an alternative strategy. The idea is to systematically sample the configuration space, test only the selected configurations, and conduct fault characterization on the resulting data. The sampling approach we use is based on calculating a mathematical object called a covering array (These are described in more detail in Section 2.1). Our experimental results show that this approach is nearly as accurate as that based on exhaustive data, but is much cheaper (provides 50-99% reductions in the number of configurations to be tested).

The remainder of this paper is organized as follows: Section 2 briefly explains the mathematical tools we used in this paper; Section 3 describes the fault characterization process; Section 4 describes the studies we conducted; Section 5 provides practical advice to users of this approach; Section 6 compares covering arrays to random selection; and and Section 7 presents concluding remarks and possible directions for future work.

2. BACKGROUND

In this paper we propose a 3-step process for characterizing faults. First we systematically sample a system's entire configuration space using a mathematical object called a covering array as opposed to using the entire configuration space as we did in [8]. Next we use Skoll to distribute and test individual configurations at remote user sites and relay the results to a central server. Finally, we classify the test results and provide the resulting model to the system's developers.

In this section we provide some background information on these three steps.

2.1 Covering Arrays

The software systems we consider in this research have options, each of which takes its value from a set of valid settings. Our main goal is to identify and characterize failures that are caused by specific combinations of options settings. Therefore, it is important that we maximize the "coverage" of option setting combinations. However, we also want to do this at some reasonably low cost. Consequently, we also want to minimize the total number of configurations tested.

Our approach to doing this is to compute t-way option coverage using a combinatorial object called a *covering array*. 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. We refer to the k columns of this array as *factors* and each of the v values for a factor as its *levels*. In the Skoll system, each of the configuration options can be modeled as one of the factors of this system. Each of the settings for these options is a level of that factor. The *strength* of the array is denoted by t. Since many software systems do not have the same number of levels for each factor we can use a *mixed level* covering array to model this system. An $MCA(N; t, k, (v_1, v_2, ...v_k))$, is an $N \times k$ array on s symbols, where $s = \sum_{i=1}^{k} v_i$. In this array each column $i \quad (1 \leq i \leq k)$ contains elements from a set S_i with $|S_i| = v_i$. The rows of every $N \times t$ sub-array cover all $t-$tuples of values from the t columns at least once. We can use a shorthand notation to describe our covering array by combining v_i's that are the same and representing this number as a superscript. For example if we have 4 $v's$ each with 3 levels, we can write this 3^4. In this manner an $MCA(N; t, k, (v_1 v_2 ... v_k))$ can also be written as an $MCA(N; t, (s_1^{p_1} s_2^{p_2} ... s_r^{p_r}))$ where $k = \sum_{i=1}^{r} p_i$.

In this paper, we restrict ourselves to mixed level covering arrays. Therefore we will use the general term *covering array* to refer to these from now on.

Covering arrays have the property that each t-tuple is used *at least* once, which means they can be arbitrarily large. One of our goals, in building these, must be to minimize the size of N. There are a variety of computational methods that can be used to find covering arrays with a small N for a given set of parameters. See [3, 4, 9, 11] for some of these methods. In [4] several greedy algorithms are compared with heuristic search such as simulated annealing and hill climbing. Simulated annealing gives a consistently small N when $t = 2$ or $t = 3$. Therefore, we chose this as our construction method. Simulated annealing is a standard combinatorial optimization technique (see [4] for a more thorough discussion of this algorithm). In this simulated annealing program, the cost function is the number of uncovered t-sets remaining, i.e. a covering array has a cost of 0. We begin with an unknown N for a particular set of parameters, repeating the annealing process many times, using a binary search strategy to find the smallest N which gives us a solution [4].

This approach has been used most frequently to test input combinations of programs. Dalal *et al.*, for example, argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults [5]. In further work, Burr *et al.*, Dunietz *et al.* and Kuhn *et al.* provide more empirical results to show that this type of test coverage is effective [2, 6, 7]. These studies focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics [3, 6].

Our approach is different in that we apply covering arrays to system configuration options and we assess their effectiveness in revealing option-related failures and finding failure inducing options.

2.2 Skoll

Skoll [8] is a distributed continuous QA process supported by automated tools which leverages the extensive computing resources of worldwide user communities in order to efficiently, incrementally and opportunistically improve software quality and to provide greater insight into the behavior and performance of fielded systems. Skoll does this by dividing global QA processes into multiple subtasks which are then intelligently distributed to client machines around the world, executed by them, and their results returned to Skoll server where they are fused together to complete the overall QA process. Figure 1 summarizes the Skoll process (refer to [8] for further details).

A corner stone of Skoll is a formal model of a QA process' configuration space called *configuration model*. The model captures configuration options and their settings as well as the constraints on them. An Intelligent Steering Agent (ISA) located at the Skoll server uses this information in planning the global QA process, for adapting the process dynamically, and to aid in interpreting the results. ISA is implemented using planning technology and utilizes various constraint solving, scheduling and planning algorithms.

In this paper, we create covering arrays for a configuration model and use Skoll to distribute and test individual configurations at remote user sites and collect the results at a central server. For a given configuration, each client downloads software from a central code repository, configures and compiles it, runs a battery of tests on it, and sends the results back to the server.

2.3 Classification Trees

Once we have obtained the test results, we use classification tree analysis to model failure-inducing options (i.e., the specific options and their settings in which the failure manifests itself).

Classification trees use a recursive partitioning approach to build a model that predicts a configuration's class (e.g., passing or failing) in terms of the values of individual option settings. This model is tree-structured. Each node denotes an option, each edge represents an option setting, and each leaf represents a class or set of classes (if there are more than 2 classes).

Classification trees are constructed using data called the *training set*. A training set consists of configurations, each with the same set of options, but with potentially different option settings together with known class information. Based on the training set, models are built as follows. First, for each option, partition the training set based on the option settings. The resulting partition is evaluated based on how well the partition separates configurations of one class from those of another. Commonly, this evaluation is realized as an entropy measure [1].

The option that creates the best partition becomes the root of the tree. To this node we add one edge for each option setting. Finally, for each subset in the partition, we repeat the process. The process stops when no further split is possible (or desirable).

To evaluate the model, we use it to predict the class of previously unseen configurations. We call these configurations the *test*

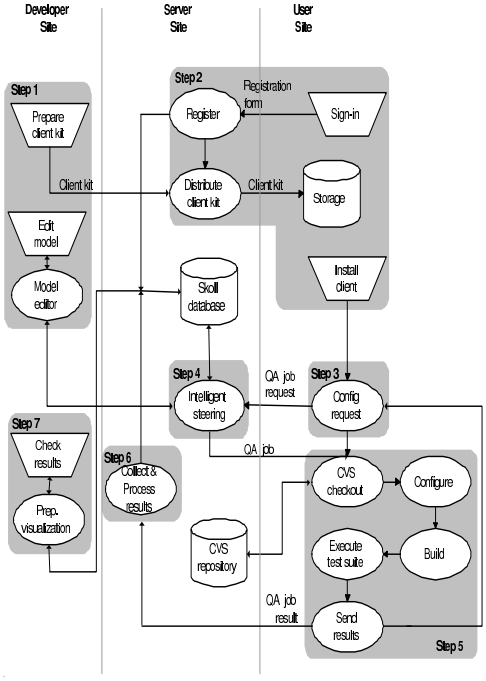


Figure 1: Process view of Skoll.

set. For each configuration we begin with the option at the root of the tree and follow the edge corresponding to the option setting found in the new configuration. This process continues until a leaf is encountered. The class label found at the leaf is interpreted as the predicted class for the new configuration. By comparing the predicted class to the actual class we estimate the accuracy of the model.

In this research, we use the classification trees to extract failure-inducing option setting patterns. That is we extract the options and option settings from the tree that characterize failing configurations. In particular we use the Weka implementation of J48 classification tree algorithm with the default confidence factor of 0.25 [10] to obtain the models.

3. THE FAULT CHARACTERIZATION PROCESS

In this research, our ultimate goal is to provide developers with compact and accurate descriptions of failing configuration subspaces. Our experience shows that such information can help developers find the causes of failures much more quickly than they can without this information [8]. In this section we provide more detail about the fault characterization process and describe how we evaluate its performance.

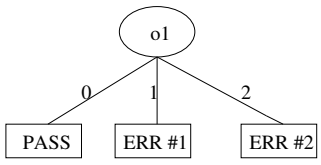


Figure 2: An example of a classification tree.

Table 1 depicts the results of exhaustively testing a system with three configuration options ($o1$, $o2$, and $o3$) each having three levels of settings (0, 1, and 2). There are no constraints among the options, so there are 27 valid configurations. These results show four outcomes – test PASSed, test failed with ERR#1, test failed with ERR #2 and test failed with ERR #3.

Config			Result	Config			Result
o1	o2	o3		o1	o2	o3	
0	0	0	PASS	1	1	2	ERR #1
0	0	1	PASS	1	2	0	ERR #1
0	0	2	ERR #3	1	2	1	ERR #1
0	1	0	PASS	1	2	2	ERR #1
0	1	1	PASS	2	0	1	ERR #2
0	1	2	PASS	2	0	2	ERR #2
0	2	0	PASS	2	0	3	ERR #2
0	2	1	PASS	2	1	0	ERR #2
0	2	2	PASS	2	1	1	ERR #2
1	0	0	ERR #1	2	1	2	ERR #2
1	0	1	ERR #1	2	2	0	ERR #3
1	0	2	ERR #1	2	2	1	ERR #2
1	1	0	ERR #3	2	2	2	ERR #2
1	1	1	ERR #1				

Table 1: An example of an exhaustive suite.

Feeding this data to a classification tree algorithm yielded the model shown in Figure 2. This simple model tells us that the setting of option $o1$ is strongly correlated with the manifestation of failures $ERR\#1$ and $ERR\#2$. That is, configurations with $o1 == 1$ fail with $ERR\#1$ and those with $o1 == 2$ fail with ERR #2.

3.1 Evaluating Fault Characterizations

Obviously, these models may not be complete and correct. In some cases this is because:

1. the underlying problem is not related to the options settings (e.g., ERR #3 occurs with all settings of $o1$ and $o2$ and 2 of the 3 settings of $o3$), or

2. the model building approach identifies spurious, but non-causal patterns.

This research is not concerned with non-option-related failures. We attempt to remove them from our analysis. In general, this can't be done without manually verifying each and every failure. Consequently, we simply remove any failure from consideration that occurs in less than 3% of the test runs. Our rationale is that deterministic failures involving up to 5 binary options should manifest at least this many times as also should non-deterministic failures involving fewer options, but appearing with a reasonable frequency (say 3 options with the failure manifesting 1/4 of the time).

To evaluate the accuracy of classification tree models we use several standard metrics. Precision (P) and recall (R) are two widely used metrics to assess the performance of classification models. For a given failure class E, they are defined as follows:

$$recall = \frac{\#\ of\ correctly\ predicted\ instances\ of\ E\ by\ the\ model}{total\ \#\ of\ instances\ of\ E}$$

$$precision = \frac{\#\ of\ correctly\ predicted\ instances\ of\ E\ by\ the\ model}{total\ \#\ of\ predicted\ instances\ of\ E\ by\ the\ model}$$

Drawing an analogy to a medical test, recall measures how well the test identifies infected people; Precision measures how many false alarms the test raises. In general we want good recall because otherwise the models may miss relevant characteristics or add irrelevant ones. On the other hand, we want to minimize false alarms

because we don't want developers to waste resources investigating them.

Because neither measure predominates our evaluation we combine the measures using the F metric. This is defined as:

$$F = \frac{(b^2+1)PR}{b^2P+R}$$

Here, b controls the weight of importance to be given to precision and recall: $F = P$ when $b = 0$ and $F = R$ when $b = \infty$. Throughout this paper, we compute F with $b = 1$, which gives precision and recall equal importance, and use it to evaluate fault characterization models.

3.2 Reducing Test Suite Size

While the model in Figure 2 explains the observed failures reasonably well, it did so at the cost of exhaustively testing the configuration space. This won't scale. Interestingly, we get the same tree model using only the shaded configurations in Table 1. Moreover, this reduced suite is only one-third the size of the exhaustive suite. We selected these configurations because they constitute a 2-way covering array of the configuration space. That is, all pairwise combinations of the options appear in the shaded configurations. If these results hold in practice, it would greatly reduce the cost of fault characterization, without compromising its accuracy. We evaluate this conjecture throughout the rest of this paper.

4. EXPERIMENTS

In this Section we describe several studies of our fault characterization approach. We applied this process to an open-source CORBA middleware implementation ACE+TAO.

ACE+TAO is a large, widely-deployed open-source middleware software toolkit that can be reused and extended to simplify the development of performance-intensive distributed software applications. The ACE+TAO source base has evolved over the past decade and now contains over one million lines of C++ source code. It is highly configurable with a large number of configuration options (over 500) supporting a wide variety of program families and standards.

In a previous study, we modeled and studied a small subset of the system's entire configuration space. This model comprises 10 compile-time and 6 runtime options. Each compile-time option was binary-valued, while the runtime options had differing numbers of settings: four options with three levels, one option with four levels, and one option with two levels. All told, this configuration space has 18,792 valid configurations.

Compile-time options allow features, such as asynchronized method invocation (AMI) and CORBA messaging, to be compiled in or out of the system. Runtime options provide more fine-grained control over the runtime behavior of the system, such as object collocation strategies and connection purging strategies.

We tested each configuration using 96 regression tests each of which were designed to emit an error message in the case of failure. The error messages were captured and recorded. In this paper, we adopt the results of these tests and refer them as *exhaustive results*.

To evaluate the use of covering arrays, we created five different t-way covering arrays for this configuration space. We allowed t to range between 2 and 6. We reran the regression tests on each of these t-way suites and used classification trees to automatically characterize the test results. We then compared the fault characterizations obtained from t-way suites to the ones obtained from exhaustive testing.

Because our earlier work uncovered numerous compilation problems, we chose to group the 10 compile time options into a single configuration with 29 levels (i.e., the 29 static configurations that compiled). Otherwise, we would have generated numerous uncompilable configurations. Our goal then became to see how well we could detect runtime errors and the failure-inducing options that lead to them. Note that this is simply a time-saving issue, it does not change the size of the underlying configuration space or the characterizations we are trying to find. Using this approach we computed an $MCA(N; t, 29^1 4^1 3^4 2^1)$. The model has seven configuration options. The first corresponds to the 29 successfully compiled static options, and the rest correspond to the 6 runtime configurations.

CA Strength (t)	No. of Tests (N)
2	116
3	348
4	1229-1236
5	3369 - 3372
6	9433-9453

Table 2: Size of test suites for $2 \leq t \leq 6$.

Table 2 gives the covering array size N for each value of t. When $t \leq 3$ all five arrays were the same size N. For these we were able to construct covering arrays with the smallest mathematically possible number of rows. When $t \geq 4$, the problem of building a small N is harder so we obtained a range of sizes.

In the remainder of the section we present the results of several studies. The first study examines how well covering array-derived testing schedules suites reveal failures. Obviously, if they don't, then fault characterizations based on them will suffer. The second study involves covering array-derived test schedules and for each test builds one characterization model for all failures observed on that test. The third study uses covering array-derived test schedules, but builds one characterization model for each observed failure on each test. Finally, the fourth study repeats study three, but uses several lower strength covering arrays, comparing them to the more expensive to obtain higher strength covering arrays.

4.1 Study 1: Revealing option-related failures with covering arrays

An initial question we had about covering arrays is whether they reveal the option related failures. If they don't, then any characterization based on them will obviously suffer.

Figure 3 plots error coverage statistics for 2-way covering arrays. The reason we include figures for only 2-way covering arrays is that they are the most interesting ones for showing the error coverage statistics since their sizes are the smallest. In this figure, each bar represents one test case. The height of a bar is the number of unique errors seen with the exhaustive test suite. The lower part of a bar (darker color) shows the average number of unique errors seen by the five 2-way suites. Tests that never failed are omitted. For example, during the execution of test #35 in the exhaustive suite we observed eight unique error messages whereas the 2-way suites only revealed three of them on the average.

As Figure 3 indicates, the 2-way suites discovered only a small percentage of the failures seen by the exhaustive suite. On the other hand, we are particularly interested in the effectiveness of the reduced suites in revealing option-related failures.

In order to identify option-related failures we used the 3% cut-off value described in Section 3.1. We removed any failure from consideration that had occurred in less than 3% of the test runs in the entire configuration space. This gave us 40 potential option-related failures. We then checked the effectiveness of covering arrays in revealing those 40 failures. It turned out that each and every t-way

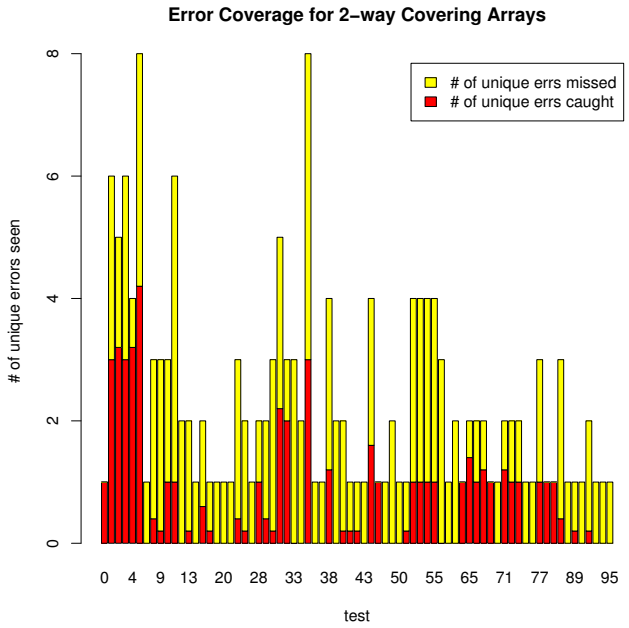


Figure 3: Error coverage statistics for 2-way covering arrays.

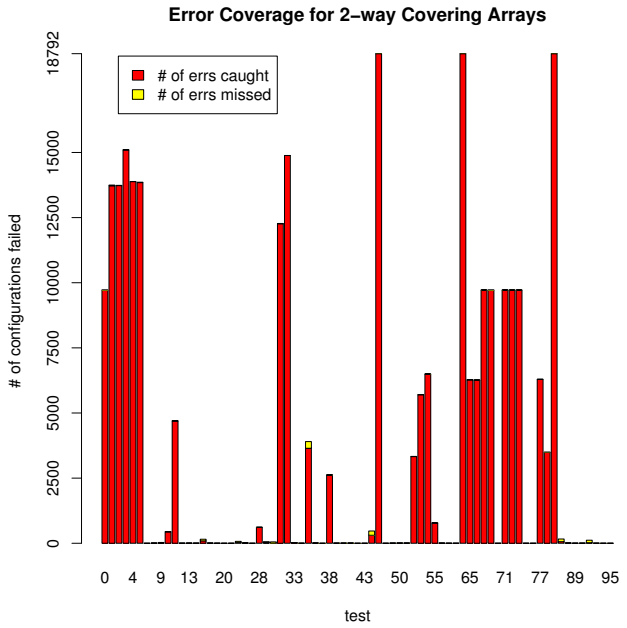


Figure 4: Error coverage statistics for 2-way covering arrays (a different view).

suite revealed all of these failures.

Figure 4 summarizes this result for the 2-way suites. This figure plots the number of failing configurations for each test. The lower part of each bar shows the number of failing configurations whose error messages are discovered by the 2-way suites and the upper part indicates the number of configurations whose error messages are missed by the 2-way suites. As can be seen from the figure, the 2-way suites discovered the failures that constitute large failing subspaces.

In the rest of the paper, we assess the effectiveness of t-way suites for fault characterization only on these potential option-related failures.

4.2 Study 2: Covering arrays with per test case characterization

In this study we use covering array-derived test schedules. For each test case, we build one characterization model for all failures observed in any of the scheduled configurations.

4.2.1 Creating classification tree models

For each configuration in the entire space, we ran all the regression tests and recorded their pass/failure information. We then built a classification model using these data and tested it on the same data. This tells us how well, in the best case, the models characterize the faults.

We repeated the process above for only the scheduled configurations (i.e., those selected by the covering array). We then built classification tree models using this data. We tested the models, however, on the exhaustive data set. This tells us how well the models, built using only a subset of the data, characterize the faults.

In the rest of the paper, we'll refer to the models obtained from the covering arrays and the exhaustive suite as reduced models and exhaustive models, respectively.

4.2.2 Evaluation

We applied the classification models to our exhaustive results and collected the F measures for each potential option-related failures.

Figure 5 shows the F measures of the reduced models and the exhaustive models for 40 potential option-related failures. The vertical axis denotes F measure, and the horizontal axis denotes test index and error index. For example, the first tick on the horizontal axis, which is 0-1, represents the error indexed as 1, which occurred during the execution of the test indexed as 0.

The first thing to note is the the F measures for the reduced models are almost always near those of the exhaustive models. That is, if the exhaustive models characterize the failure well, then so do the reduced models. If they don't, then neither do the reduced models. This is true no matter what the strength of the covering array (the level of t) is. For example, 78% of the models obtained from the 2-way schedules gave F measures within 0.1 of the exhaustive models. 88% of them were within 0.2. The higher the strength of the covering arrays, the closer the F measures were. Also important, is the fact that the 2-way covering arrays provided this similar performance while providing a 99.4% reduction in the number of configurations to be tested! In our experiments with ACE+TAO, for example, it took us 8 hours to compile ACE+TAO, compile the tests, and execute them for each configuration. Using 2-way suites would have saved us almost a year of machine time compared to using the exhaustive suite without dramatically lowering the accuracy of the fault characterizations.

Our analysis also suggests that the higher the F measure, the more similar the exhaustive and reduced models were in terms of the options and settings captured. To make the analysis clearer we divided the models into four categories using the F measures obtained from exhaustive models: very strong ($F = 1$), strong ($0.8 < F < 1$), weak ($0 < F \leq 0.8$), and unknown ($F = 0$).

The reduced models for very strong patterns were exactly the same as the exhaustive models. That is, they produced the same sets

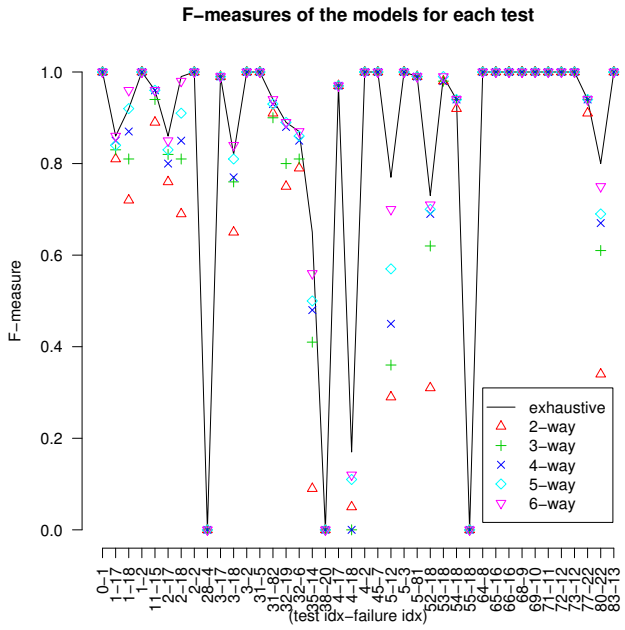


Figure 5: Models for each test.

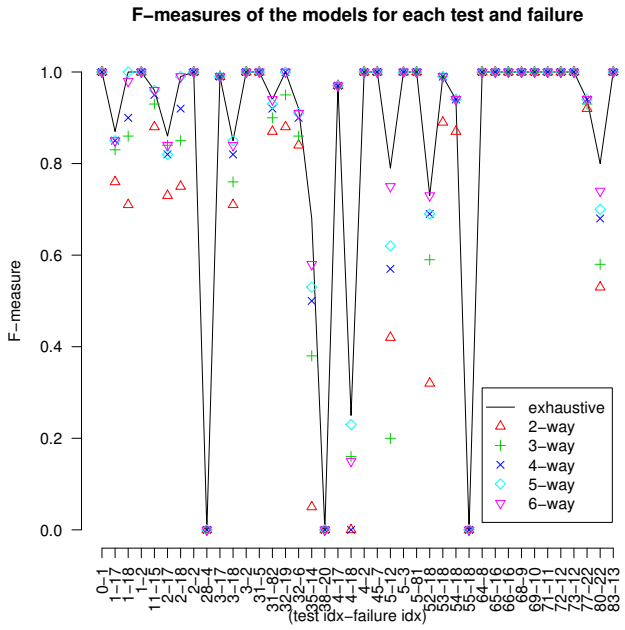


Figure 6: Models for each test and failure combination.

of rules to describe the failures. The similarity decreased steadily as we moved to strong patterns and then to weak patterns. Weak patterns captured by the reduced models (especially the 2-way models) tended to differ substantially from those found in the exhaustive models – See failures 52-18, 80-22, and 35-14. In these cases we saw that using higher strength covering arrays boosted performance.

Failures with unknown patterns are interesting. Although these failures were seen frequently enough to be considered potential option-related failures, the classification model found no apparent pattern to their occurrences. We observed three failures in this nature, namely 28-4, 38-20, and 55-18. None of the suites, even the exhaustive suite, were able to provide fault characterizations for these failures.

This result amplifies our earlier results [8] in which we showed that the patterns contained in the classification trees often, but not always, corresponded with the actual cause of the failure. As we will show in Section 5, the concept of pattern strength gives us a way to determine whether the classification tree model is reliable, and, therefore, likely to help developers find an actual failure cause.

4.3 Study 3: Covering arrays with per test, failure case characterization

Building classification models with several classes can lead to situations where there is too little data from which to conclude class assignment or to situations where global model building choices lead to suboptimal models for individual classes.

In this study we attempt to circumvent this problem by using covering array-derived test schedules, building one characterization model for each test case and failure combination.

4.3.1 Creating classification tree models

Just as in Study 2, we ran all test cases on every configuration in the configuration space and recorded their pass/failure information. For each test and failure f we created a training data set. Here we recoded the test outcomes into two classes: those failing with failure f and those passing. We repeated the process with the covering array-derived schedules and compared the results.

4.3.2 Evaluation

Figure 6 shows the F-measures for the models. At a first glance this performance is indistinguishable from that of Study 2. Consequently, our findings in Study 2 apply also to this study.

One important way in which these two approaches differ however is in the readability of the resulting models. When we build one model for multiple failures, as we did in Study 2, extraneous information can creep into the patterns that describe the different failures.

Figure 7(a), (b) and (c) illustrate this situation. Figure 7(a) shows the characterization for two failures that occurred during the execution of test #3 (we've excluded other errors to simplify the discussion). This model says that error #2 occurs when CALLBACK==0 and that error #17 occurs when CALLBACK==1 and ORBCollocation==NO. We know from earlier analysis, however, that error #2 occurs when CALLBACK==0 and that error #17 occurs when ORBCollocation==NO. That is, the setting of CALLBACK has no effect on the manifestation of error #17. That the CALLBACK option appears in the pattern for error #17 is simply an artifact of the modeling process when there are multiple classes being modeled.

When we build a model for each test and failure combination, on the other hand, this problem doesn't appear. In fact, the fault characterizations, shown in Figures 7(b) and (c), are exact and are the actual causes of the failures.

Error #2 occurred during the compilation of the test case. It turned out that certain files within TAO implementing CORBA messaging incorrectly assumed that CALLBACK option would always be set to 1. Consequently, when CALLBACK==0 certain definitions were unset.

Error #17 occurred when the ORBCollocation optimization was turned off. ACE+TAO's ORBCollocation option controls the con-

```
CALLBACK=0:ERR #2
CALLBACK=1
| ORBCollocation=glb:PASS
| ORBCollocation=orb:PASS
| ORBCollocation=NO:ERR #17
```

(a)

```
CALLBACK=0:ERR #2
CALLBACK=1:PASS
```

(b)

```
ORBCollocation=glb:PASS
ORBCollocation=orb:PASS
ORBCollocation=NO:ERR #17
```

(c)

Figure 7: Fault characterizations for test #3, test #3 and error #2, and test #3 and error #17, respectively.

ditions under which the ORB should treat objects as being collocated. Turning it off means that objects should never be treated as being collocated. When objects are not co-located they call each other's methods by sending messages across the network. When they are collocated, they can communicate directly, saving networking overhead. The fact that these tests worked when objects communicated directly, but failed when they talked over the network clearly suggested a problem related to message passing. In fact, the source of the problem was a bug in their routines for marshaling/unmarshalling object references.

As the strength of the covering arrays increased, fault characterizations become closer to the ones obtained from the exhaustive suite. We illustrate the differences among the characterizations obtained from different strength covering arrays in Figure 8.

Figure 8(a), (b), and (c) show the fault characterizations obtained from exhaustive suite, 2-way covering arrays, and 3-way covering arrays, respectively for error #18 which occurred during the execution of test #3. The exhaustive model correlated the failure with four options and gives an F measure of 0.849. The 2-way model was able to link the failure to only one option. This resulted in an F measure of 0.747. On the other hand, the 3-way model associated the failure with three options and resulted in a better F measure, (0.795), than the 2-way model.

4.4 Study 4: Combined reduced suites

As shown in Table 2, the size of the covering array schedules grows rapidly as t increases. In this study we examined how combined lower strength schedules compare to single higher strength covering arrays (e.g., 3, 2-way covering arrays vs. 1, 3-way covering array).

Specifically, we combined schedules in such a way that the size of the combined t-way schedules is close to the size of a single $(t+1)$ schedule. We then compared the combined schedules to the uncombined ones. This is interesting because the cost of creating $(t+1)$-way suites can be significantly higher than the cost of obtaining t-way suites (the cost is exponential in t). If t-way-combined and $(t+1)$-way suites have comparable performance measures then using the combined suites can be cost-effective.

4.4.1 Creating classification tree models

We created combined t-way schedules by randomly selecting from uncombined t-way schedules. No duplicates were allowed. We created 5 combined schedules for t from 2 to 5. We didn't combine 6-way suites because the average size of the 6-way suites was almost half that of the exhaustive suite. The average sizes of the t-way-combined suites are given in Table 3.

Classification models were built as in Study 3.

```
POLLER=0
| DIOP=0
| | INTERCEPTOR=0
| | | MUTEX=0:PASS
| | | MUTEX=1:ERR #18
| | INTERCEPTOR=1:ERR #18
| DIOP=1
| | INTERCEPTOR=0:ERR #18
| | INTERCEPTOR=1
| | | MUTEX=0:ERR #18
| | | MUTEX=1:PASS
POLLER=1:PASS
```

(a)

```
POLLER=0:ERR #18
POLLER=1:PASS
```

(b)

```
POLLER=0
| MUTEX=0
| | INTERCEPTOR=0:PASS
| | INTERCEPTOR=1:ERR #18
| MUTEX=1:ERR #18
POLLER=1:PASS
```

(c)

Figure 8: Fault characterizations for error #18 obtained from exhaustive suite, 2-way covering arrays, and 3-way covering arrays, respectively.

Suite	Size
2-way-combined	344.20
3-way-combined	1357.60
4-way-combined	3450.60
5-way-combined	8422.00

Table 3: Size of combined suites.

4.4.2 Evaluation

Figure 9 plots the F measures for t-way and t-way-combined suites. t-way-combined suites resulted in better fault characterizations compared to the t-way suites. In particular, they boosted the characterizations of failures where single suites gave low F measures (i.e. less than 0.5).

For example, consider the 2-way and 2-way-combined models for test #35, error #14 shown in Figure 9. The F measures for these models are 0.06 and 0.39, respectively. The combined suite gives an F measure that is much closer to that of the 3-way suite, which is 0.42. On the other hand, when the F measures of single suites are already high (say greater than 0.5), the combined suites don't improve performance to a great degree.

One possible explanation for this improvement is that the combined suites cover 82-89% of the t+1 tuples. Thus, they provide many of the data points seen in the t+1 covering arrays, but at a much lower construction cost.

5. GUIDELINES FOR SOFTWARE PRACTITIONERS

We have evaluated our fault characterization process by comparing it to the results of exhaustive testing. In practice, developers will not have access to this information. Therefore, in this section, we provide some guidelines on how to use this approach in practice.

In particular, we examine how to interpret reduced models, how to estimate whether the reduced models are reliable, how to select the appropriate strength level for the covering arrays, and how to

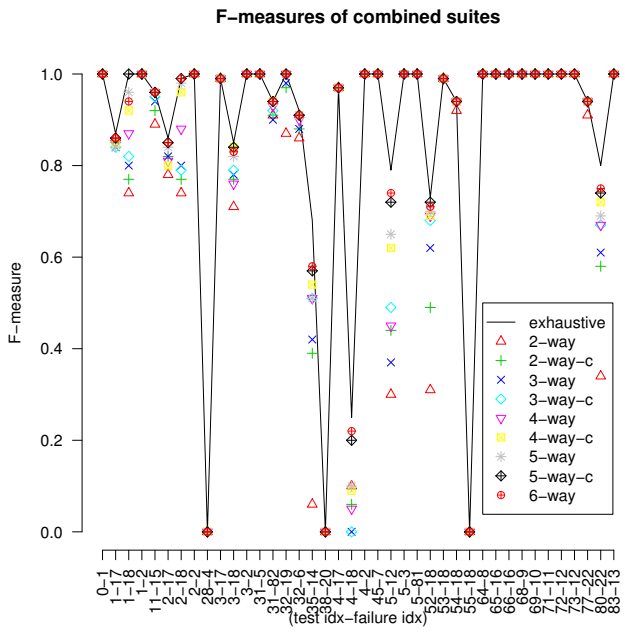


Figure 9: Models for combined suites.

work with a set of models.

Classification tree models can be partially evaluated without a traditional test set. Typically this is done using a k-fold stratified cross-validation strategy [10]. Assuming that $k == 10$, for example, the training data is randomly divided into ten parts. Within each part the classes should be represented in approximately the same proportions as in the original data set.

Then for each of the 10 parts, a model is built using the remaining nine-tenths of the data and tested to see how well it predicts for that part. Finally, the ten error estimates are averaged to obtain an overall error rate. A high error rate indicates that the models are highly sensitive to the subset of the data with which they are constructed. This suggests that the models may be "overfit" and shouldn't be trusted.

We performed stratified ten-fold cross-validation on our reduced models from Study 3. Across the 40 different failures we found that whenever the reduced model's cross-validation F measures were 0, the failure was either very rare (not considered option-related) or was one for which even the exhaustive model couldn't find a fault characterization (i.e., $F = 0$). These failures were, namely 28-4, 38-20, and 55-18. This suggests that models with 0 F measures are unlikely to signal option-related failures.

As a next step, we investigated the relation between the cross-validation F measures and the F measures of the exhaustive models. Figures 10(a) and (b) depict scatter plots of these two F measures for the 2-way and the 4-way models, respectively. We show only two figures due to space limitations. The trends of the other models are similar. We see the two F measures are very similar (they lie near the x=y line). The higher the strength of the arrays, the closer the F measures are.

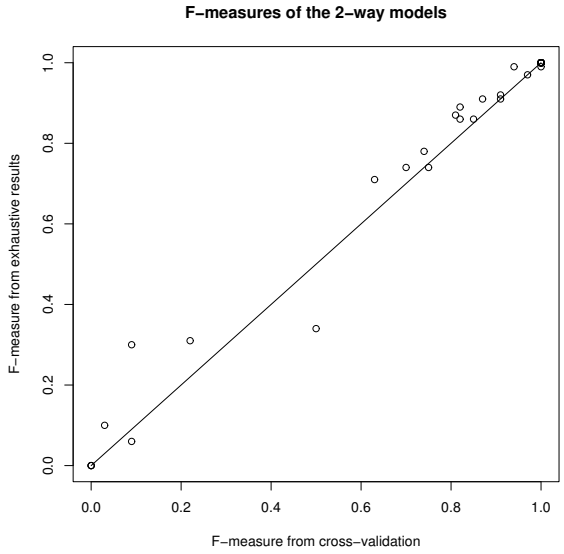
This suggests that F measures from the cross-validation of reduced models can help estimate the performance of the models when they are applied to the exhaustive results.

Based on the findings above, we give the following guidelines to the users of covering arrays:

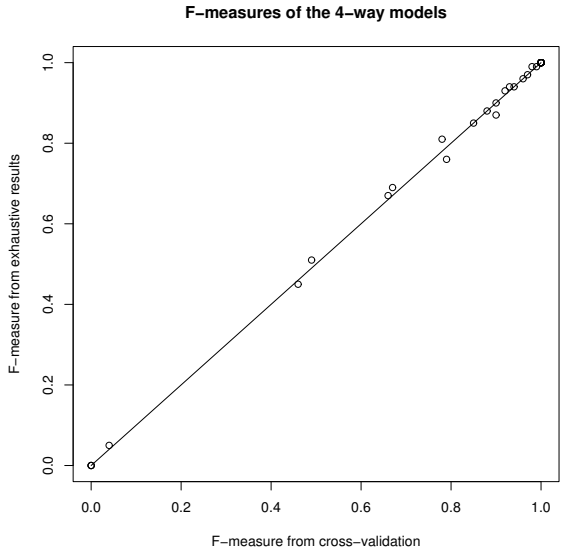
1. Use the F measures obtained from cross-validations of reduced models to flag unreliable models.

2. Higher F values are more likely to signal accurate fault characterizations, which in turn can help pinpoint the causes of failures quickly and accurately. Investigate the models with the highest F-measures first.

3. Consider using higher strength covering arrays or combined ones for the failures whose F values are low (i.e., less then 0.5).



(a)



(b)

Figure 10: Scatter plots of F measures for 2-way and 4-way models.

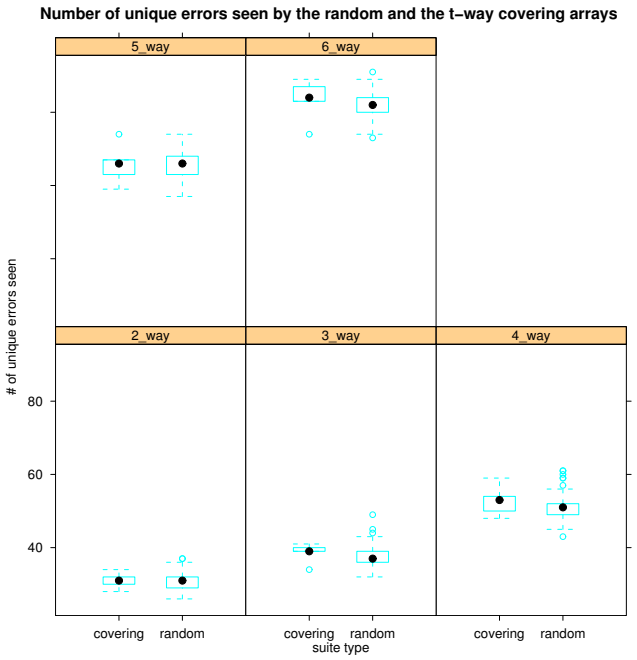


Figure 11: Number of unique errors seen in random and t-way covering suites.

6. COMPARISON WITH RANDOM SUITES

In this section, we compare the effectiveness of t-way and randomly selected schedules. For this, we created 100 random suites for each value of t where the size of each random schedule is the same as the corresponding t-way schedule.

Our first concern was to see how well the random schedules revealed failures. Figure 11 contains boxplots for the number of failures observed by the random and t-way schedules conditioned on t. In general we see that the higher the value of t (and thus the larger its size), the greater the number of failures observed. The t-way suites tend to reveal slightly more failures than the corresponding random suites with less variance.

Next we evaluated the two scheduling approaches in terms of their fault characterizations. For this, we randomly chose 15 randomly selected schedules for each value of t and created the classification tree models for option-related failures. In general, we observed that random and t-way schedules yielded comparable fault characterization models.

Random schedules, however, were worse in situation where they completely missed option-related failures and where they resulted in unbalanced sampling of the failing subspaces. In the first situation, obviously, the models ignored the failure because it had not been observed when running the random schedule. The second situation occurs when some parts of the configuration space are tested much more frequently than others. This often lead to spurious options to be included in the models.

Figure 12 illustrates this situation by contrasting the fault characterizations for test #2, ERR #18 obtained from the exhaustive schedule, a 2-way schedule, and a random schedule. The F measures for the models are 0.993, 0.774, and 0.436, respectively. The exhaustive schedule gave the model shown in Figure 12(a). Compare this to the 2-way schedule appearing in Figure 12(b). The latter is simpler and thus incorrect in some cases because it doesn't recognize the importance of the MUTEX option. Still, it doesn't include any unrelated options that would distract a developer trying to find the cause of the failure.

```
POLLER=0
| MUTEX=0
| | INTERCEPTOR=0:PASS
| | INTERCEPTOR=1:ERR #18
| MUTEX=1
| | INTERCEPTOR=0:ERR #18
| | INTERCEPTOR=1:PASS
POLLER=1:PASS
```

(a)

```
POLLER=0:ERR #18
POLLER=1:PASS
```

(b)

```
POLLER=0
| ConnectStrategy=0:PASS
| ConnectStrategy=1:ERR #18
| ConnectStrategy=2:PASS
POLLER=1:PASS
```

(c)

Figure 12: Fault characterization for test #2, ERR #18 obtained from the exhaustive suite, a 2-way suite, and a random suite, respectively.

The model created from the random schedule however (Figure 12(c)) includes a node for the ConnectionStrategy option right under the node for the POLLER option. Our analysis shows that this option is unrelated to the underlying failure. This happened because, with the random schedule, when POLLER == 0, 86% of the configuration with ConnectionStrategy == 1 fail with ERR #18. Thus, to the model building algorithm ConnectionStrategy == 1 appears to be important in explaining the underlying failure. In contrast, in the exhaustive and 2-way schedules only 21% and 33% of the configurations with ConnectionStrategy == 1 fail. This difference is simply due to a "unlucky" random selection that produced an unbalanced sampling of the underlying configuration space.

In summary, we observed that random and t-way schedules gave comparable fault characterizations on the average, but that the random schedules sometimes created unreliable models. Moreover, in practice, the covering array approach automatically determines the size of the schedule, whereas there's no way to predetermine the correct size of a randomly selected schedule.

7. CONCLUSION

Fault characterization in configuration spaces can help developers quickly pinpoint the causes of failures, hopefully leading to much quicker turn-around time for bug fixes. Therefore, automated techniques, which can effectively, quickly, and accurately perform fault characterization, can save a great deal of time and money throughout the industry. This is especially true where system configuration spaces are large, the software changes frequently, and resources are limited.

To make the process more efficient, we recast the problem of selecting test schedules (determining which configurations to test) as a problem of calculating a t-way covering array over the system configuration space. Using this schedule, we ran tests and fed the results to a classification tree algorithm to localize the observed faults. We then compared the fault characterizations obtained from exhaustive testing to those obtained via the covering array-derived schedule.

- We observed that building fault characterizations for each observed fault rather than building a single one for all observed

faults led to more reliable models.

- We observed that even low strength covering arrays, which provided up to 99% reduction in the number of configurations to be tested, often had fault characterizations that were as reliable as those created through exhaustive testing.
- Higher strength covering arrays performed better than lower strength ones and yielded more precise fault characterizations, but were more costly.
- We also showed that we could improve the fault characterization accuracy at low cost by combining lower strength covering arrays rather than increasing the covering array strength.

We were also able to develop some diagnostic tools to support software practitioners who want to use covering arrays in fault characterizations. In particular we found that:

- Low F measures in the exhaustive models tended to be associated with overfit models or non-option-related failures. These models are not likely to help developers identify option-related failures.
- We found that the F measures taken from 10-fold cross-validation were highly correlated and nearly identical with those taken from exhaustive models. This suggests that that cross-validation measures, which can be taken without having already done exhaustive testing, might be a useful surrogate for the exhaustive model F measures.

In continuing work, we are integrating covering arrays calculations directly into the Skoll system. At the same time the Skoll system is being integrated into the daily build process of several large-scale, widely used systems such as ACE+TAO. This will give us a chance to replicate the experiments over much larger and more realistic configuration spaces. We are also examining how to better model the effect of inter-option constraints on the fault characterizations.

8. REFERENCES

[1] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

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

[3] 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):437–44, 1997.

[4] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '03)*, pages 38–44, 2003.

[5] 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 *Proc. of the Intl. Conf. on Software Engineering, (ICSE)*, pages 285–294, 1999.

[6] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '97)*, pages 205–215, 1997.

[7] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[8] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. *To be appear in Proc. of the Intl. Conf. on Software Engineering, (ICSE '04)*, 2004.

[9] K. C. Tai and L. Yu. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[10] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[11] T. Yu-Wen and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Proc. of IEEE Aerospace Conf.*, pages 431–7, 2000.