Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces

Cemal Yilmaz, Myra B. Cohen, *Member*, *IEEE*, and Adam A. Porter, *Senior Member*, *IEEE*

Abstract—Many modern software systems are designed to be highly configurable so they can run on and be optimized for a wide variety of platforms and usage scenarios. Testing such systems is difficult because, in effect, you are testing a multitude of systems, not just one. Moreover, bugs can and do appear in some configurations, but not in others. Our research focuses on a subset of these bugs that are "option-related"—those that manifest with high probability only when specific configuration options take on specific settings. Our goal is not only to detect these bugs, but also to automatically characterize the configuration subspaces (i.e., the options and their settings) in which they manifest. To improve efficiency, our process tests only a sample of the configuration space, which we obtain from mathematical objects called covering arrays. This paper compares two different kinds of covering arrays for this purpose and assesses the effect of sampling strategy on fault characterization accuracy. Our results strongly suggest that sampling via covering arrays allows us to characterize option-related failures nearly as well as if we had tested exhaustively, but at a much lower cost. We also provide guidelines for using our approach in practice.

Index Terms—Software testing, distributed continuous quality assurance, fault characterization, covering arrays.

1 Introduction

Many modern software systems must be customized to specific runtime 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) have dozens, even hundreds, of options. While this flexibility promotes customization, it creates many potential system configurations, each of which may need extensive quality assurance (QA). We call this problem *software configuration space explosion*. To address this issue, we have developed Skoll [13]—a distributed continuous QA (DCQA) process supported by automated tools that 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.

One QA process implemented in Skoll determines which specific options and option settings cause specific failures to manifest. We call this process *fault characterization*. We do it by testing different configurations and feeding the results to a predictive model-building process [13]. The output models describe the options and settings that best predict failure. For example, for a Corba implementation, we determined that when the executable ran on the Linux operating system with Corba Messaging Support enabled but with Asynchronous Message Invocation support disabled, socket connections frequently timed out.

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. The fault characterization, however, greatly narrowed down the search space, making the developers' job much easier.

While we were pleased with this outcome, the approach requires us to test the entire configuration space. In the example cited above, this means that nearly 19,000 times, remote clients spent several hours downloading, configuring and sometimes compiling the 2M+ LOC system and then executing numerous tests. And this was only a small subset of the system's much larger configuration space. Clearly, a more efficient process is needed.

In earlier work, we proposed and evaluated an alternative strategy [19]. Our idea was to cut testing costs by systematically sampling the configuration space, testing only the selected configurations, and conducting fault characterization on the resulting data. The sampling approach we used is based on a mathematical object called a covering array (described in more detail in Section 2.1). Covering arrays induce a test schedule that ensures that all t-way interactions between options are observed at least once. Our evaluation showed that this approach was nearly as accurate as that based on exhaustive data, but was much less expensive. (It provided a 50 to 99 percent reduction in the number of configurations tested.) This paper extends that earlier work in two ways. First, we replicate and expand the

- *C. Yilmaz and A. Porter are with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: {cyilmaz, aporter}@cs.umd.edu.*
- *M. Cohen is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, NE 68588-0115. E-mail: myra@cse.unl.edu.*

Manuscript received 12 July 2005; revised 17 Nov. 17 2005; accepted 2 Dec. 2005. Published online XX Xxx. 2006.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0198-0705.

TABLE 1
A Covering Array Example $CA(9;2,3,3)$

Configuration No	Option A	Option B	Option C
1	0	0	0
2	0	1	1
3	0	2	2
4	1	0	1
5	1	1	2
6	1	2	0
7	2	0	2
8	2	1	0
9	2	2	1

original study by including a second operating-system environment. Second, we introduce and evaluate the use of a new kind of covering array, called a variable-strength covering array (described in Section 2.2), which provides developers with finer control over covering array construction. The remainder of this paper is organized as follows: Section 2 briefly explains the mathematical tools we use in this paper, Section 3 describes the fault characterization process, Sections 4 and 5 describe the studies we conducted, Section 6 provides practical advice to users of this approach, Section 7 compares covering arrays to random selection, Section 8 discusses related work, and Section 9 presents concluding remarks and possible directions for future work.

2 Background

In this paper, we use a three-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 testing the entire configuration space as we did in earlier work [13]. Next, we test individual configurations at remote user sites, which relay the results to a central server. Finally, we classify the test results and provide the resulting models to the system's developers. Now, we provide some background information on these three steps.

2.1 Covering Arrays

The software systems considered in this research have *options*, which take their values from a set of valid *settings*. Our goal is to identify and characterize failures that are caused by specific combinations of option settings. Therefore, it is important to maximize the "coverage" of option-setting combinations. However, since we also want to keep costs low, we must also minimize the number of configurations tested. The set of configurations to be tested is called the *test schedule*. To do this, we compute 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 any $N \times t$ subarray contains all ordered t-sets of size v at least once [4]. The *strength* of the array is denoted by t. For instance, given a covering array of strength $t = 2$ we can arbitrarily select any two columns from the covering array to form a new subarray. We are guaranteed that any ordered pair from the v values will be found in at least one row of this subarray. When using the Skoll system, each of the configuration options is a column of the covering array. Each option setting is mapped to one of the v values for that column. This gives us a covering-array-derived test schedule, or *CA test schedule*. A CA test schedule for a configuration space is a set of N test configurations in which all t-way combinations of option settings appear at least once.

TABLE 2
A Mixed-Level Covering Array Example: $(12;2,3^2 4^1)$

Configuration No	Option A	Option B	Option C
1	0	1	1
2	0	2	0
3	2	1	0
4	0	0	2
5	1	1	2
6	2	2	2
7	2	2	3
8	2	0	1
9	1	0	0
10	1	2	1
11	1	1	3
12	0	0	3

Consider the following system with three ternary options, A, B, and C, each with the possible settings 0, 1, and 2. This system has 27 possible configurations. A $CA(9;2,3,3)$ for this system is shown in Table 1. As promised, for any two columns, all possible pairs of option settings can be found.

When software systems have options with varying numbers of settings, we must use a *mixed-level* covering array. An $MCA(N;t,k,(v_1,v_2,\ldots v_k))$, is an $N \times k$ array on s symbols, where $s = \sum_{i=1}^{k} v_i$. In this array, each column i $(1 \leq i \leq k)$ contains elements from a set S_i with $|S_i| = v_i$. The rows of every $N \times t$ subarray cover all t-tuples of values from the t columns at least once. A shorthand notation is used to describe a covering array by combining v_is that are the same and representing this number as a superscript. For example, if we have 4 v_is, each with three values, this can be written as 3^4. In this manner, an $MCA(N;t,k,(v_1 v_2 \ldots v_k))$ can also be written as an $MCA(N;t,(s_1^{p_1} s_2^{p_2} \ldots s_r^{p_r}))$, where $k = \sum_{i=1}^{r} p_i$.

Returning to the previous example, suppose option C now has four possible settings instead of three. We can create a mixed-level covering array using 12 configurations (shown in Table 2). Here, all possible pairs of the four settings for option C are combined with the three settings for options A and B. The combinations of all three settings from options A and B are all accounted for as well. This is an $MCA(12;2,3^2 4^1)$. In this paper, we will always use mixed-level covering arrays, which we will refer to as *covering arrays* for simplicity.

In general, we want our covering arrays to be as small as possible. A variety of computational methods exist for finding small covering arrays for a given set of parameters. Simulated annealing is a standard combinatorial optimization technique (see [7]) that has been shown to consistently provide small covering arrays when $t = 2$ or $t = 3$. Therefore, we chose to use this construction method. In our implementation of the simulated annealing method, 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 that gives us a solution [7].

TABLE 3
A VSCA Example: $VSCA(10;2,3^3 2^3, CA(10;3,3,2))$

Config. No	Option A	Option B	Option C	Option D	Option E	Option F
1	2	2	1	1	0	1
2	0	2	2	0	0	1
3	2	1	2	0	1	0
4	1	0	2	1	1	1
5	0	2	0	1	1	0
6	0	1	1	1	1	1
7	0	0	1	0	1	1
8	2	0	0	0	0	0
9	1	1	0	0	0	1
10	1	2	1	1	0	0

2.2 Variable-Strength Covering Arrays

Covering arrays define a "fixed" t across all of the k columns. In [6], [7] an aggregate object called a *variable-strength covering array* is defined. A *variable-strength covering array* is a covering array of strength t with subsets of columns of strength $> t$. It is denoted as a $VSCA(N;t,(v_1,v_2,..,v_k),C)$. More formally, it is an $N \times k$ mixed-level covering array of strength t containing C, a vector of covering arrays, each of strength $> t$ and defined on a subset of the k columns.

This structure provides the ability to tune a test schedule so that certain sets of options are tested more strongly (i.e., higher strength for certain option groups) while maintaining t-way coverage across the whole system. This can be useful when it is too expensive to increase coverage across all options or when developers know that some option groups are more likely to cause faults or cause more serious faults. Conversely, sometimes, a variable-strength test schedule can be created that is the same size as a covering-array test schedule. This occurs when there is a large imbalance in the numbers of option settings across the system. We take advantage of this situation in some of the studies in this paper. Suppose we have a system with three ternary options (A, B, and C) and three binary options (D, E, and F). Further, suppose that the binary options control interrelated functionality and are therefore known to interact. In this case, we might want to exhaustively test the three binary options (which requires at least eight configurations). We could use a three-way covering array for the whole system, but this requires at least 27 configurations since the first three options, contain three settings each. If, however, we are happy with a minimum of two-way coverage overall, we can build the VSCA shown in Table 3. These 10 configurations include all possible combinations of options D, E, and F, while also covering all possible two-way combinations between *any* of the six options. It is a $VSCA(10;2,3^3 2^3, CA(10;3,3,2))$.

We construct variable-strength covering arrays using simulated annealing [6]. The cost function here is the missing t-sets added to the sum of the missing tuples for all covering arrays in the vector C.

2.3 Skoll

To improve the quality of software systems with complex configuration spaces, we are exploring (DCQA) processes [13] that evaluate various software qualities, such as portability, performance characteristics, and functional correctness, "around-the-world, around-the-clock." To accomplish this, a general DCQA process is divided into multiple subtasks, such as running regression tests on one particular system configuration, evaluating system response time under one of several input workloads, or measuring usage errors for a system with one of several alternative GUI designs. These subtasks are then intelligently and continuously distributed to—and executed by—clients across a grid of computing resources. The results of these evaluations are returned to servers at central collection sites, where they are fused together to guide subsequent iterations of the DCQA processes contributed largely by end-users and distributed development teams. To support this effort we have developed *Skoll*, an infrastructure for designing and executing DCQA processes. Its components and services include languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and planning technology that analyzes subtask results and adapts the DCQA process in real time. See [13], [18], [19] for more details.

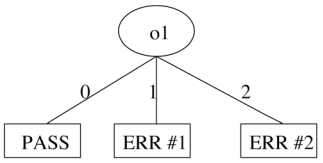


Fig. 1. An example classification tree.

2.4 Classification Trees

We use *classification tree analysis* (CTA) to model failing configuration subspaces [1]. CTA is a recursive partitioning approach to build models that predict a configuration's class (e.g., passing or failing) based on the settings of the options that define a configuration. This model is tree-structured (see Fig. 1). Each node denotes an option, each edge represents a possible option setting, and each leaf represents a class or set of classes (if there are more than two 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.

1. For each option, partition the training set based on the settings of that option.
2. Evaluate the option based on how well it partitions configurations of different classes.
3. Select the best option and make it the root of the tree.
4. Add one edge to the root for every option setting.
5. Repeat the process for each new edge. 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 (called the *test 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. We continue until a leaf is encountered. The leaf's class label is then 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 analyze the classification trees to extract failure-inducing option setting patterns, i.e., the set of options and their settings that characterize failing configurations. We use the Weka implementation of the J48 classification-tree

TABLE 4
An Example Exhaustive Test Schedule

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	0	ERR #2
0	1	2	PASS	2	0	1	ERR #2
0	2	0	PASS	2	0	2	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				

algorithm with the default confidence factor of 0.25 [17] to build classification-tree models.

3 The Fault-Characterization Process

Our goal is to give developers compact, accurate descriptions of failing configuration subspaces. We've found that such information can help developers quickly narrow down the causes of failure [13]. This section details our fault-characterization process and its evaluation.

Table 4 shows exhaustive testing of a system with three ternary configuration options ($o1$, $o2$, and $o3$), each of which has settings (0, 1, and 2). There are no interoption constraints, so there are 27 valid configurations. In this example, we observed four outcomes: test PASSed, test failed with ERR #1, test failed with ERR #2, and test failed with ERR #3.

Applying CTA to this data yields the classification tree model shown in Fig. 1. This model tells us that configurations with $o1 == 1$ fail with ERR #1 and those with $o1 == 2$ fail with ERR #2.

3.1 Evaluating Fault Characterizations

In practice, classification trees may not be perfectly accurate. Reasons for this might include: 1) The failure is unrelated to the option settings. For example, in our earlier example, ERR #3 occurs in configurations having all settings of $o1$ and $o2$ and two of the three settings of $o3$; or 2) the model-building approach identifies spurious, noncausal patterns.

This research focuses on option-related failures. Therefore, we try to remove nonoption-related failures from our analysis. Since we can't do this automatically, we simply ignored all failures that occurred in less than 3 percent of the test runs. Our rationale was that deterministic failures involving up to five binary options should manifest at least this many times. The same is true of nondeterministic failures involving fewer options but appearing with a reasonable frequency (e.g., failures involving three options with the failure manifesting 1/4 of the time).

To evaluate the classification-tree models, we used standard metrics: precision (P) and recall (R). For a given failure class E:

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

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

Recall measures how well the model predicts configurations that experience failure E. Precision measures how many configurations are falsely identified as experiencing failure E. In general, both measures are important. We want high recall because otherwise the models may miss relevant characteristics or add irrelevant ones. And we want high precision to minimize wasting resources while investigating false alarms.

Since neither measure predominates our evaluation, we combine the measures using the F metric [14]:

$$F = (b^2 + 1)PRb^2P + R.$$

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.

3.2 Reducing the Test Schedule Size

While the model in Fig. 1 explains the observed failures well, we had to exhaustively test the configuration space to get it. Since this won't scale, we need a way to build the models based on data taken from only a subset of the entire configuration space. Interestingly, we could have derived the same tree model using data from only 1/3 of the configuration space (the configurations boxed in Table 4). These selected configurations, in fact, constitute a two-way covering array of the configuration space. (This is the same covering array depicted in Table 1). If similar results occur in practice, fault characterization would be much cheaper, without compromising accuracy. We examine this conjecture in the following sections.

4 Experiments

This section presents several studies of our CA-based fault-characterization approach. Our goal is to compare the costs and benefits of the modified approach to those of the original approach which requires testing the entire configuration space. Our subject program for these studies is the ACE+TAO system [15], [16]. ACE+TAO is a large, widely deployed open-source middleware software toolkit. The ACE+TAO source base contains over 2 million lines of C++ source code. It is highly configurable with over 500 configuration options supporting a variety of program families and standards.

In a previous study [13], we applied our original fault-characterization process to a subset of the system's entire configuration space. That subset consisted of 10 compile-time and six runtime options. Each compile-time option is binary-valued and allows various features to be compiled in or out of the system. In addition, there are 12 interoption constraints that restrict the total number of compile-time configurations. The system's runtime options have differing numbers of settings—i.e., four

TABLE 5
Size of Test Schedules for $2 \leq t \leq 6$

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

options have three settings, one option has four, and one option has two. Runtime options control runtime optimizations, set system-level policies, and generally provide fine-grained control over the runtime behavior of the system.

All told, this subset of the system had over 53,000 valid configurations. Note, however, that we uncovered numerous compilation problems during testing. These static configurations were excluded from the model, which reduced the configuration space to 18,792 valid configurations.

We tested each compilable, valid configuration on the Red Hat Linux 2.4.9-3 platform and on Windows XP Professional using 96 developer-supplied regression tests. Each test was designed to emit an error message in the case of failure, and we captured and recorded the results of each test. This testing took over two machine years to run. In the rest of the paper, we refer to this data set as the *exhaustive results*.

To evaluate using covering arrays, we created five different t-way covering arrays for each value of t ranging between 2 and 6. Specifically, we computed an $MCA(N; t, 29^1 4^1 3^4 2^1)$ for each value of t. Because of the numerous compile-time errors we uncovered earlier, we chose to group the 10 compile-time options into a single 29-valued option. That is, we mapped each of the valid 10-option strings to a single value setting. Thus, the model has seven configuration options. The first corresponds to the 29 successfully compiled static configurations, and the rest correspond to the six runtime options.

We reran the regression tests for each of these t-way test schedules on both platforms and used classification trees to automatically characterize the test results. We then compared the fault characterizations obtained from t-way schedules to those obtained from exhaustive testing.

Table 5 gives the covering array size N for each value of t. When $t \leq 3$, all five arrays were the same size. For these, we were able to construct covering arrays with the smallest mathematically possible size. When $t \geq 4$, the problem of building a minimally-sized covering array becomes harder, so we obtained a range of sizes.

In the remainder of this section, we present the results of four studies, each examining a different aspect of the CA-based fault-characterization process. The first study examines how well CA test schedules reveal option-related failures. The second study uses CA test schedules and builds one characterization model for each test (pass versus fail). The third study uses CA test schedules but builds one characterization model for each observed failure on each test (pass versus failure-1 versus failure-2, etc.). Finally, the fourth study repeats the third, but compares using the combination of several lower-strength covering arrays to using one (more expensive to obtain) higher-strength covering array.

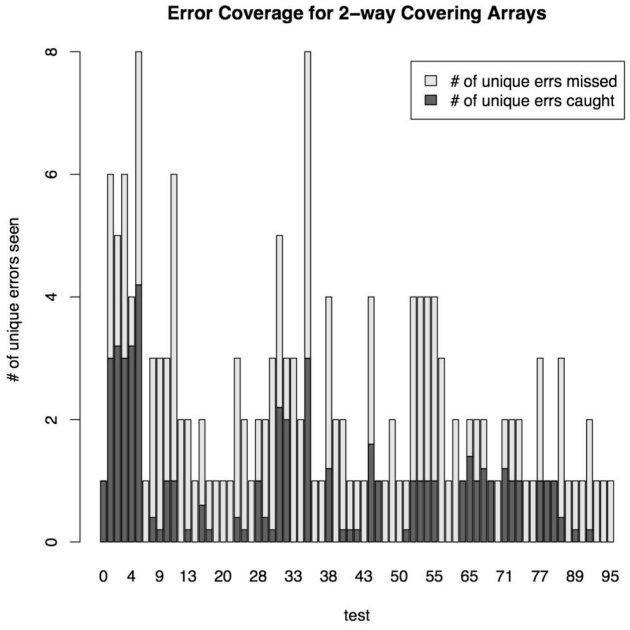


Fig. 2. The error coverage statistics for two-way covering arrays on Linux.

4.1 Study 1: Revealing Option-Related Failures with Covering Arrays

The first question we examine is whether testing only the CA test schedule negatively affects fault detection. If it does, then later fault characterization will surely suffer.

Fig. 2 plots error-coverage statistics for two-way covering arrays on the Linux platform. We show data only for two-way covering arrays as they are the smallest. In this figure, each bar represents one test case. Tests that never fail are omitted. The height of a bar represents the number of unique error messages observed with the exhaustive test schedule. The lower part of a bar (darker color) shows the average number of unique errors observed by the five two-way schedules. For example, using the exhaustive schedule, we observed eight unique error messages while running test #35. Using the two-way schedules, however, we only observe three of them on average. The Windows platform shows similar results.

Fig. 3 provides another view of this data. Instead of the number of unique error messages, it depicts the number of configurations in which each test fails. The lower part of each bar shows the average number of failing configurations whose error messages are detected at least once by the two-way schedules. The upper part indicates the average number of failing configurations whose error messages are not detected by the two-way schedules. As suggested by the figure, failures not detected by the two-way schedules occur with very low frequency. Alternatively, the CA schedules are able to detect all faults that appear with reasonable frequency. Since we are interested in characterizing option-related failures, we aren't overly concerned with rarely occurring failures. This is because rarely occurring failures are either 1) likely not related to option settings; if they were they would appear more frequently, or 2) unlikely to be accurately characterized even with exhaustive testing, e.g., a failure that occurs exactly once in 20,000 configurations does not allow for statistical generalization.

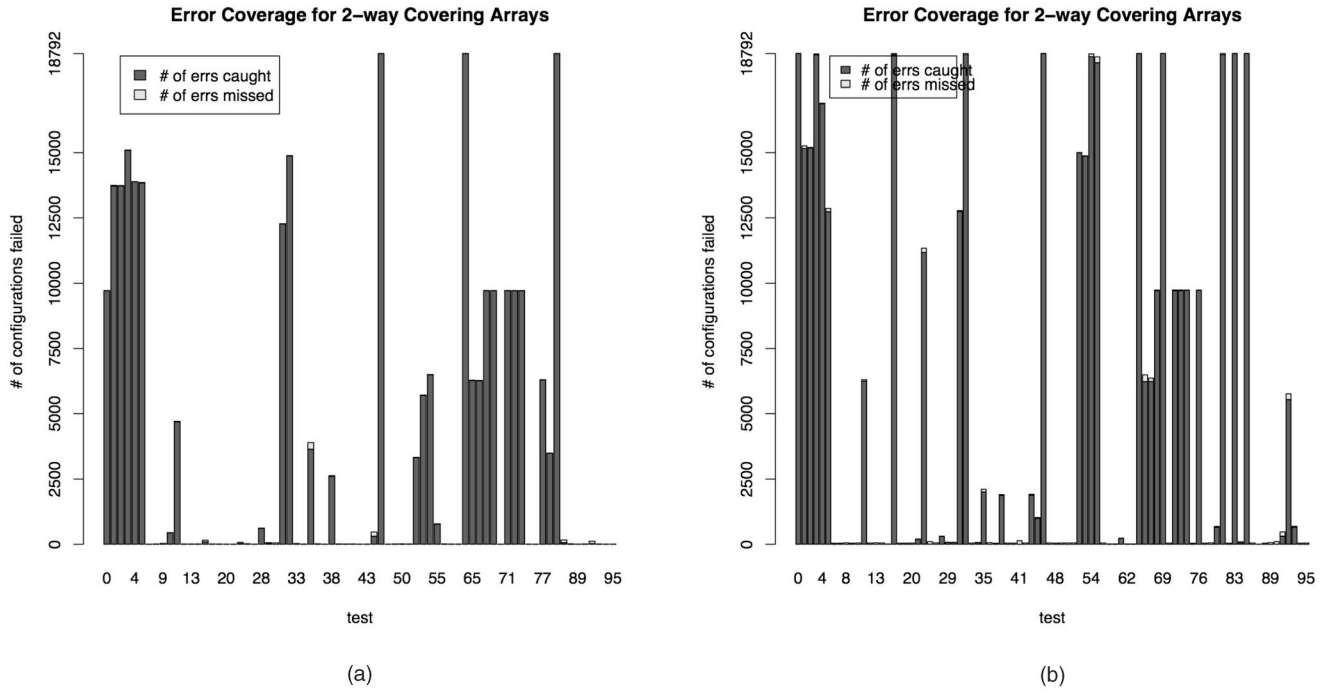


Fig. 3. Error coverage statistics for two-way covering arrays. (a) Linux. (b) Windows.

As mentioned in Section 3.1, we consider a failure to be potentially option-related only if it appears in more than 3 percent of the original configuration space. This gives us 40 "potentially" option-related failures on the Linux platform and 49 on the Windows platform. We will refer to these as the *option-related failures*. We then check the effectiveness of covering arrays in revealing option-related failures. Each and every t-way schedule reveals all option-related failures on both platforms.

4.2 Study 2: Covering Arrays with Per-Test-Case Characterization

The previous study suggests that testing with CA schedules reveals potentially option-related failures as well as exhaustive testing does. Given this assurance, we now compare fault characterization based on CA schedules to that based on exhaustive testing.

4.2.1 Creating Classification-Tree Models

To address this question, we run all test cases on the exhaustive schedule. For each test case, this results in a set of passing configurations and f sets of failing configurations, one for each unique observed failure. For each test case, we then build one model that characterizes all $f + 1$ possible outcomes. This provides an upper bound on classification accuracy.

We then repeat the process using just the configurations selected by the covering array. We then test the models on the exhaustive data set. Finally, we examine how well the models built using only a subset of the data compare to those built using all of the data. We refer to the models obtained from the covering arrays as *reduced models* and those from the exhaustive schedule as *exhaustive models*.

4.2.2 Evaluation

Fig. 4 shows the F measures for the reduced models and for the exhaustive models for the 89 option-related failures. The vertical axis denotes the F measure, and the horizontal axis denotes the test and error index. For example, the first tick on the horizontal axis, which is 0-1, represents the first error observed in test case 0.

The figure suggests that 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 independent of the strength of the covering array. For example, on Linux, 78 percent of the models obtained from the two-way schedules gave F measures within 0.1 of the exhaustive models; 88 percent of them were within 0.2. The higher the strength of the covering arrays, the closer the F measures were. Another interesting observation is that the two-way covering arrays achieve this performance while reducing the number of configurations to be tested by 99.4 percent. Using two-way schedules would therefore have saved almost two years of machine time, without substantially 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 are in terms of the model rules (specific options and settings captured within the models). To do this analysis, we first pair the exhaustive and reduced models for each test case. We then divide the pairs of models into four categories based on the *strength* of the F measures of the exhaustive models: very high ($F = 1$), high ($0.8 < F < 1$), moderate ($0 < F \leq 0.8$), and low ($F = 0$).

For the very high F-measure group, the paired models are exactly the same (except for the two-way models for failures 80-262 and 93-361 on Windows). That is, the exhaustive and reduced models contain the same rules to

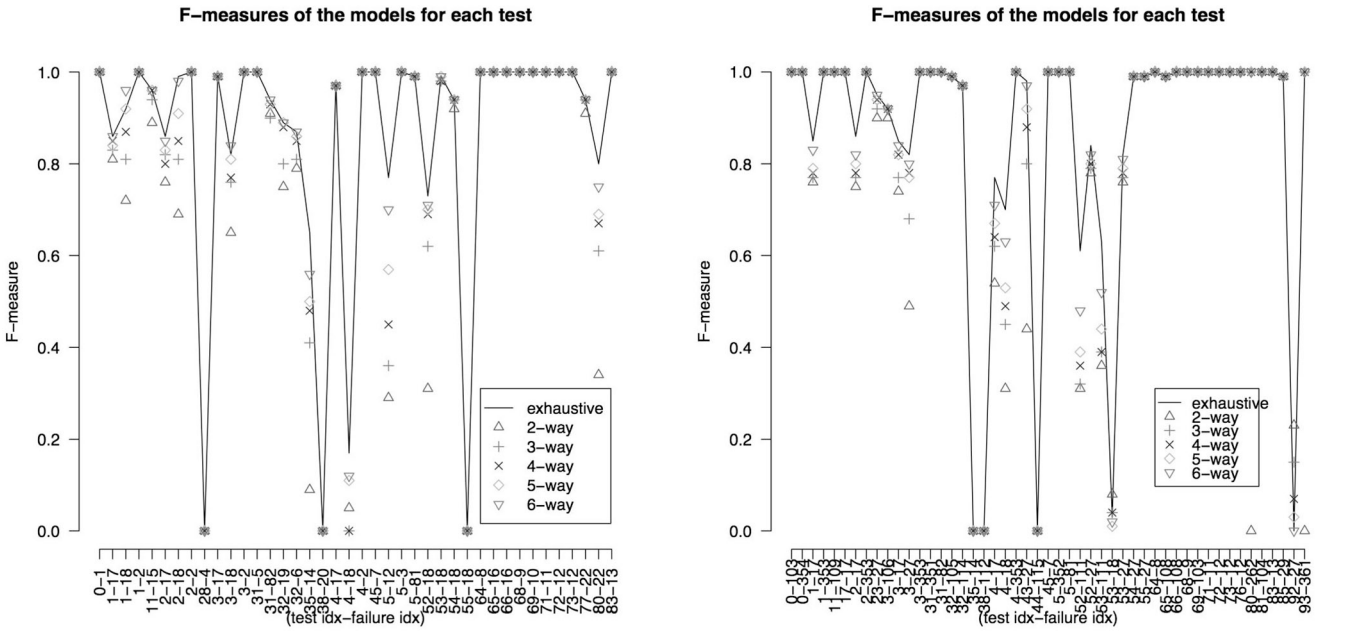


Fig. 4. Models for each test. (a) Linux. (b) Windows.

describe the failures. The two exceptions are failures that manifested in relatively few configurations (i.e., the number of failing configurations in the entire space is very near the 3 percent cutoff). Consequently, the two-way schedules observe the failures in very few configurations (i.e., four failing configurations on average), which negatively affects the resulting fault-characterization models. The similarity between paired models decreases steadily as we move down to the high and moderate F-measure groups. In the moderate F-measure group, the rules captured by the reduced models (especially the two-way models) tend to differ substantially from those captured by the exhaustive models (See failures 52-18, 80-22, and 35-14 in Fig. 4a). In these cases, we see that using higher-strength covering arrays boosted performance. The low-F measure group comprises models that fail to find any accurate pattern to the failures. Since no accurate pattern is found, the reduced and the exhaustive models may find different but equally inaccurate patterns.

These results confirm and amplify our initial study [19]. They suggest that covering-array schedules can generate data that is capable of accurately characterizing the options and option settings in which option-related faults manifest. Moreover, as we will show later, the concept of pattern strength will help us predict when classification-tree models are likely to be 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 on which to base class assignment and to situations where global model-building choices lead to suboptimal models for individual classes. In this study, we try to avoid this by building one characterization model for each test-and-failure combination.

4.3.1 Creating Classification-Tree Models

Just as in Study 2, we run all test cases on every configuration in the configuration space and record their pass/failure information. For each test and failure f, we create a training data set but record only two test outcomes: failing with failure f and passing. We repeat the process with the CA schedules and compare the results.

4.3.2 Evaluation

Fig. 5 shows the F measures for the models. At first glance, they are indistinguishable from Study 2. One important way in which they 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. Fig 6 illustrates this situation. Fig. 6a shows the characterization for two failures that occurred during the execution of test #3 on Linux (we've excluded other errors to simplify the discussion). This model says that ERR #2 occurs when CALLBACK==0 and that ERR #17 occurs when CALLBACK==1 and ORBCollocation==NO. Although this seems like a reasonable classification, it is slightly inaccurate.

ERR #2 occurs during the compilation of the test case. Certain files within TAO implementing CORBA messaging incorrectly assume that the CALLBACK option would always be set to 1. Consequently, when CALLBACK==0, certain definitions are unset.

ERR #17 occurs when the ORBCollocation optimization is turned off. ACE+TAO's ORBCollocation option controls the conditions 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 collocated, they call each other's methods by sending messages across the network. When they are collocated,

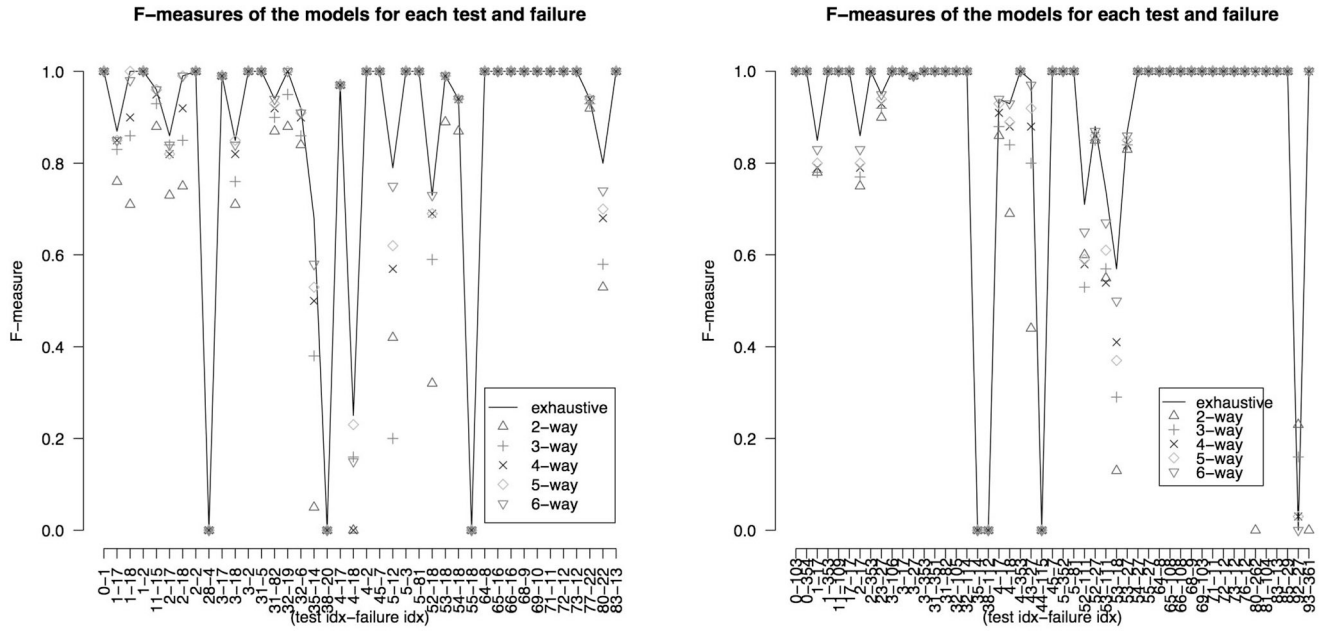


Fig. 5. Models for each test and failure combination. (a) Linux. (b) Windows.

they can communicate directly, saving networking overhead. The fact that these tests work when objects communicate directly but fail when they talk over the network clearly suggests a problem related to message passing. In fact, the source of the problem was a bug in the routines for marshaling/unmarshaling object references.

Returning to Fig. 6a, we know 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. The appearance of the CALLBACK option in the pattern for error #17 is an artifact of the modeling process when there are multiple classes being modeled together. When we remove this coupling and build a separate model for each test-and-failure combination, this problem doesn't appear. In fact, the fault characterizations, shown in Figs. 6b and 6c, exactly capture the failures' causes.

Using this per-test, per-failure characterization, we find that as the strength of the covering arrays increases, fault characterizations move closer to those obtained from the exhaustive schedule. We illustrate this in Fig. 7.

Figs. 7a, 7b, and 7c show the fault characterizations obtained from the exhaustive schedule, two-way covering arrays, and three-way covering arrays, respectively, for error #18, which occurred during the execution of test #3 on Linux.

The exhaustive model correlates the failure with four options and gives an F measure of 0.849. The two-way model is able to link the failure to only one option. This results in an F measure of 0.747. On the other hand, the three-way model associates the failure with three options and resulted in a better F measure, (0.795), than the two-way model.

4.4 Study 4: Combined Reduced Schedules

As shown in Table 5, the size of the CA test schedules grows rapidly as t increases. The cost to create them does as well (the cost is exponential in t). In this study, we examine how combined lower-strength schedules compare to single higher-strength covering arrays (e.g., three two-way covering arrays versus one three-way covering array).

Specifically, we combine 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 compare the combined schedules to the uncombined ones. This is interesting because the cost of creating $(t+1)$-way schedules can be significantly higher than the cost of obtaining t-way schedules. If t-way-combined and $(t+1)$-way schedules have comparable performance measures, then using the combined schedules can be cost-effective.

```
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)

Fig. 6. Fault characterizations for test #3, test #3 and error #2, and test #3 and error #17, respectively.

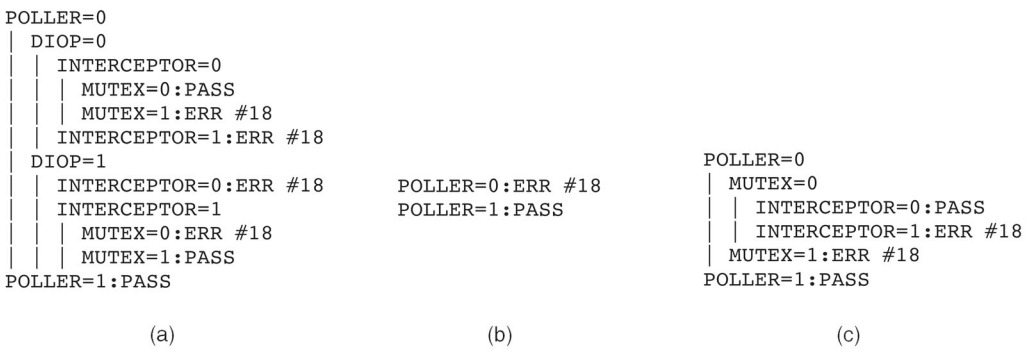


Fig. 7. Fault characterizations for error #18 obtained from the exhaustive schedule, two-way covering arrays, and three-way covering arrays, respectively.

4.4.1 Creating Classification-Tree Models

We create combined t-way schedules by merging randomly selected uncombined t-way schedules. No duplicate test configurations are allowed. We create five combined schedules for t from two to five. We don't combine six-way schedules because the average size of the six-way schedules is almost half that of the exhaustive schedule. The average sizes of the t-way-combined schedules are given in Table 6. Classification models are built as in Study 3.

4.4.2 Evaluation

Fig. 8 plots the F measures for t-way and t-way-combined schedules. t-way-combined schedules result in better fault characterizations than the t-way ones but do not do quite as well as the $t+1$ ones. In particular, the combined schedules boost the characterizations of faults when single lower-strength schedules give low F measures (i.e., less than 0.5).

For example, consider the two-way, two-way-combined (two-way-c), and three-way models for test #35, error #14 shown in Fig. 8a. The F measures for these models are 0.06, 0.39, and 0.42, respectively. The combined schedule gives an F measure that is much closer to that of the three-way schedule. On the other hand, when the F measures of single schedules are already high (say, greater than 0.5), the combined schedules don't improve performance to a great degree.

One possible explanation for the closeness in results between the $t+1$ and combined schedules is that the combined schedules cover 82-89 percent of the $t+1$-tuples. Thus, they provide many of the data points seen in the $t+1$ covering arrays, but at a lower construction cost.

5 Further Improving the Efficiency

Our current sampling strategy is based on computing a t-way covering array that covers all t-way combinations of option settings. This sampling strategy, by fixing t—the strength of the array—across the entire configuration space, treats configuration spaces as flat spaces; each t-way combination of option settings is considered equally likely to cause failures. However, our experience shows that configuration spaces are often composed of several subspaces each with, potentially, a different level of risk of causing failures. For example, in ACE+TAO, we see that faults tend to be concentrated all in static options or all in runtime options, not generally a mix of both. Testing higher-level interactions in high-risk subspaces while keeping a relatively low-level interaction coverage in the overall space can improve the efficiency of the fault characterization process. As discussed earlier, variable-strength covering arrays (VSCA) provide a method for doing just this (see Section 2.2).

We hypothesize that VSCAs can improve the efficiency of the fault characterization process in two ways: 1) They can reduce the cost of the process without compromising its accuracy by only testing the required set of high-level interactions, or 2) for the same cost, they can improve the accuracy of the process by testing more interactions. We evaluate this hypothesis in the rest of this section.

TABLE 6
Size of Combined Schedules

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

5.1 Creating Variable-Strength Covering Arrays

We have created several VSCAs for our configuration model given in Section 4. Our strategy was to use higher-strength coverage between only the runtime options. The reason behind this strategy is two-fold. First, we observed that a significant fraction of the failures we saw involved runtime options. Therefore, testing higher-level interactions between runtime options can improve the characterization models for these faults without compromising the others. Secondly, the overriding factor in the size of our covering array is the single static option; in the covering-array model we used, the 10 compile-time options were grouped into a single option with 29 settings, whereas the runtime options have at most 4 option settings. Leveraging this fact by individually manipulating the configuration space of the runtime options allows us to create VSCAs with two levels of strength (i.e., the highest level of strength is assigned to the runtime options) and with overall sizes very close to that of our fixed-strength covering arrays. This provides a way to reliably evaluate the performance boost due to VSCAs by comparing them to similar-sized CAs.

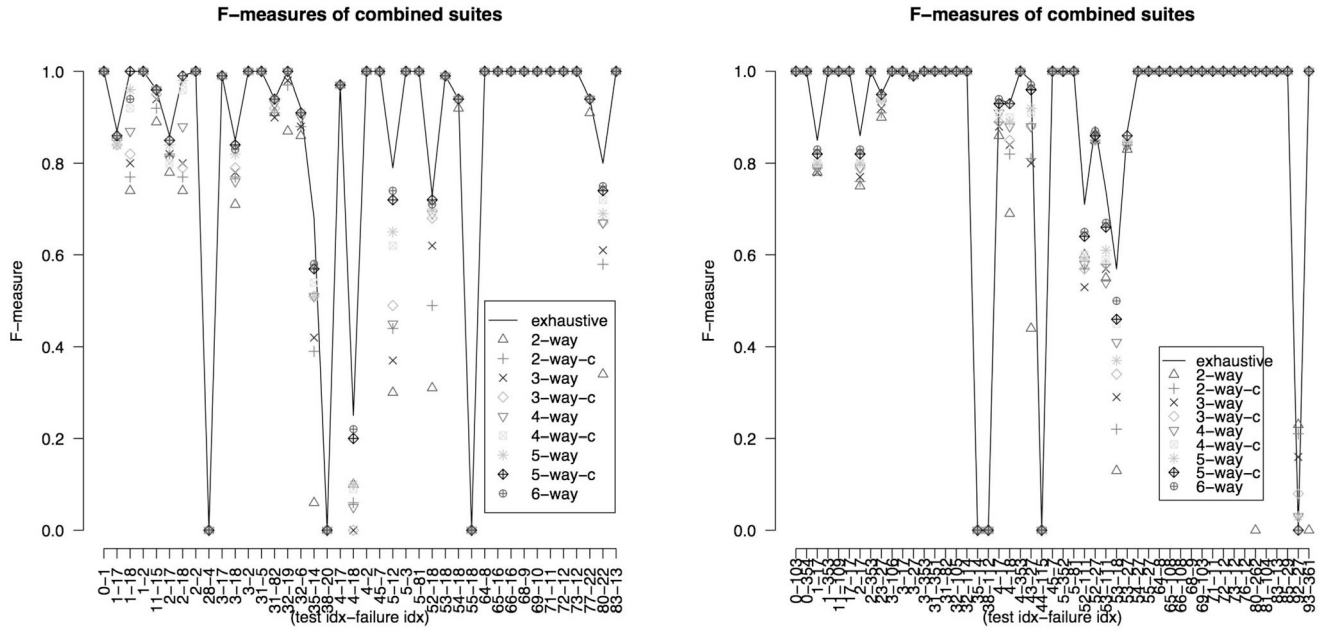


Fig. 8. Models for combined schedules. (a) Linux. (b) Windows.

We created our VSCAs with the highest level of strength that could be obtained for all of the runtime options, that would create a VSCA very close to the size of one of our covering arrays with fixed strength. The first two VSCAs created have a base strength of 2. In the first one, a $VSCA(N; 2, 29^1 4^1 3^4 2^1, MCA(N; 4, 4^1 3^4 2^1))$, the six runtime options have strength 4. The size of this VSCA is 116, which is exactly the same as the fixed-strength covering array with $t = 2$. We call this array the "two-way-overall-four-way-runtime" (abbreviated as the "2c4r" array). The second VSCA created has $t = 5$ for all of the runtime options ($VSCA(N; 2, 29^1 4^1 3^4 2^1, MCA(N; 5, 4^1 3^4 2^1))$. This VSCA has 324 configurations, which is slightly less than the three-way fixed-strength array (348 configurations). This array is called the "two-way-overall-five-way-runtime" array (abbreviated as "2c5r"). The third VSCA, $((VSCA(N; 3, 29^1 4^1 3^4 2^1, MCA(N; 5, 4^1 3^4 2^1))$, has a base strength of $t = 3$, while the six runtime options are of strength $t = 5$. This is called the "three-way-overall-five-way-runtime," (abbreviated "3c5r"). This array has 367-368 configurations, which is comparable with the size of the three-way arrays.

By creating two-way-overall-four-way-runtime, two-way-overall-five-way-runtime, and three-way-overall-five-way-runtime test schedules, we expect to improve the efficiency of the process in characterizing faults that are caused by interaction of four or more runtime options.

5.2 Evaluating Variable-Strength Covering Arrays

As in Section 4, we compute five different schedules for each of two-way-overall-four-way-runtime, two-way-overall-five-way-runtime, and three-way-overall-five-way-runtime arrays. We ran all test cases on every configuration selected by these VSCAs and recorded their pass/failure information. We created fault-characterization models for each test and failure as described in Section 4.3 and then compared the resulting models to those of fixed-strength covering arrays where $t = 2, 3, 4$.

Table 7 compares the F measures of characterization models obtained from VSCAs and CAs for some failures caused by runtime option settings. In this table, we observe that 1) the 2c4r schedules improve the fault characterizations over the same-sized two-way schedules, 2) the 2c5r schedules, compared to the three-way schedules, result in comparable—in most cases better—characterizations while providing a 6 percent reduction in the number of configurations to be tested, and 3) the fault-characterization models obtained from 3c5r schedules are always better than those of three-way schedules. These results are encouraging but not conclusive. This is because there are very few cases in which we can observe the performance improvement due to VSCAs. The fixed-strength schedules, even the low-strength ones (e.g., two-way and three-way schedules), almost always result in perfect models (F=1) for faults caused by interactions of runtime options. This may suggest that there are a limited number of faults involving four or more runtime options in our experimental data, which would prevent us from evaluating VSCAs properly. To investigate further, we run a clean-room experiment where we seed faults, the frequencies of the failures, and the

TABLE 7
Comparing Fault-Characterization Models Obtained from VSCAs and CAs Using F Measures

Failure	OS	2-way	2c4r	2c5r	3-way	3c5r	4-way
2-17	Linux	0.78	0.81	0.83	0.81	0.83	0.81
80-22	Linux	0.34	0.51	0.65	0.61	0.65	0.67
4-18	Windows	0.69	0.79	0.83	0.84	0.85	0.88

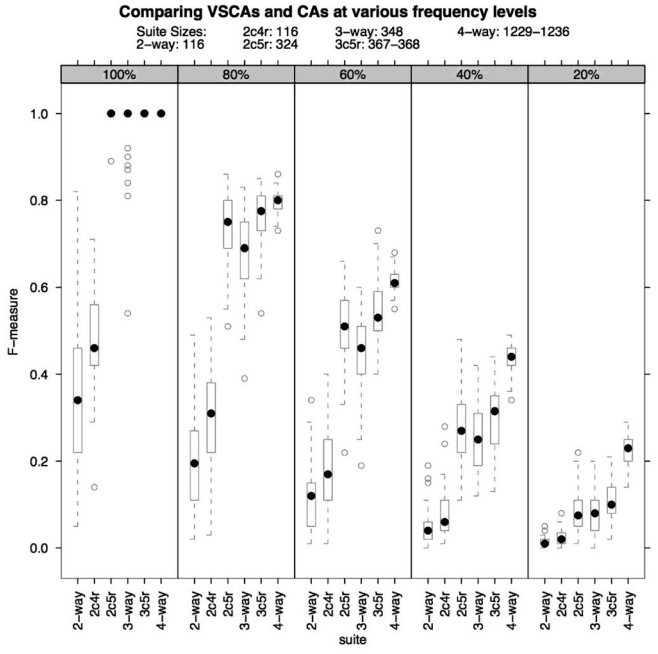


Fig. 9. Comparing VSCAs with CAs at various frequency levels.

options that are responsible for the manifestation of the faults into the system.

5.3 Seeding Faults

To evaluate the performance boost due to the VSCAs, we seed faults into the existing configuration space that are caused by simultaneous interactions of four runtime options.

We randomly select four runtime options from our configuration space (one option with two levels of setting and three options with three levels of setting each). We then seed a unique fault for each combination of these runtime option settings. This gives us 54 unique four-way faults. For each fault, we then create a separate test case, failing deterministically only on configurations in which the right combination of option settings is met. We repeat the same processes to seed faults with various occurrence frequencies (i.e., 80 percent, 60 percent, 40 percent, and 20 percent). At an x percent occurrence frequency, failures manifest themselves only at x percent of the configurations in which the failing conditions are met. For each occurrence frequency, we run all 54 hypothetical test cases on the fixed-strength and variable-strength schedules, and record their pass/failure information. We compute the fault-characterization models using the Weka ld3 algorithm [17][1] for each test and failure and then compare the resulting models.

Fig. 9, grouped by the occurrence frequency, shows the distributions of the F measures from the fixed and variable-strength schedules. We see that the VSCAs result in better characterization models when compared with their fixed-cost counterparts (sizewise), i.e., two-way versus 2c4r, three-way versus 2c5r, and three-way versus 3c5r. The differences were more pronounced at the 80 percent level. For instance, at the 100 percent occurrence, the 2c4r provides a better classification, but once a larger subset of configurations are included, all of the characterization models do equally well. At the 80 percent level, all of the VSCAs show improvement over their CA counterparts. For instance, the 2c5r improves over the three-way CA even though the 2c5r contains slightly fewer configurations. The performance differences gradually begin to diminish, however, as the level of occurrence frequencies drop below 80 percent.

1. The ld3 algorithm performs better than the J48 on small training-data sets.

6 Guidelines for Software Practitioners

We have evaluated our fault-characterization process by comparing it to exhaustive testing. In practice, developers will not have this information. Therefore, we provide preliminary 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, how to vary the strength across the configuration spaces, and how to work with a set of models.

We begin by describing an analysis method that we apply to our experimental results and then give guidelines for fixed-strength covering arrays based on this analysis. We then provide guidelines for variable-strength covering arrays based on our experience.

Classification-tree models can be partially evaluated without a traditional test set. Typically, this is done using a k-fold stratified cross-validation strategy [17]. Assuming that k==10, for example, the training data is randomly divided into 10 parts. Within each part, the classes should be represented in approximately the same proportions as in the original data set. Next, 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 that part. Finally, the 10 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 perform stratified 10-fold cross-validation on our reduced models from Study 3. We only present the analysis results obtained from the Linux experiments. The results for the Windows experiments are similar. We find that whenever the reduced model's cross-validation F measures are 0, the failure is either very rare (not considered option-related) or is an option-related failure for which even the exhaustive model couldn't find a fault characterization (i.e., $F = 0$). These failures are, 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 investigate the relation between the cross-validation F measures and the F measures of the exhaustive models. Figs. 10a and 10b depict scatter plots of these two F measures for the two-way and the four-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.

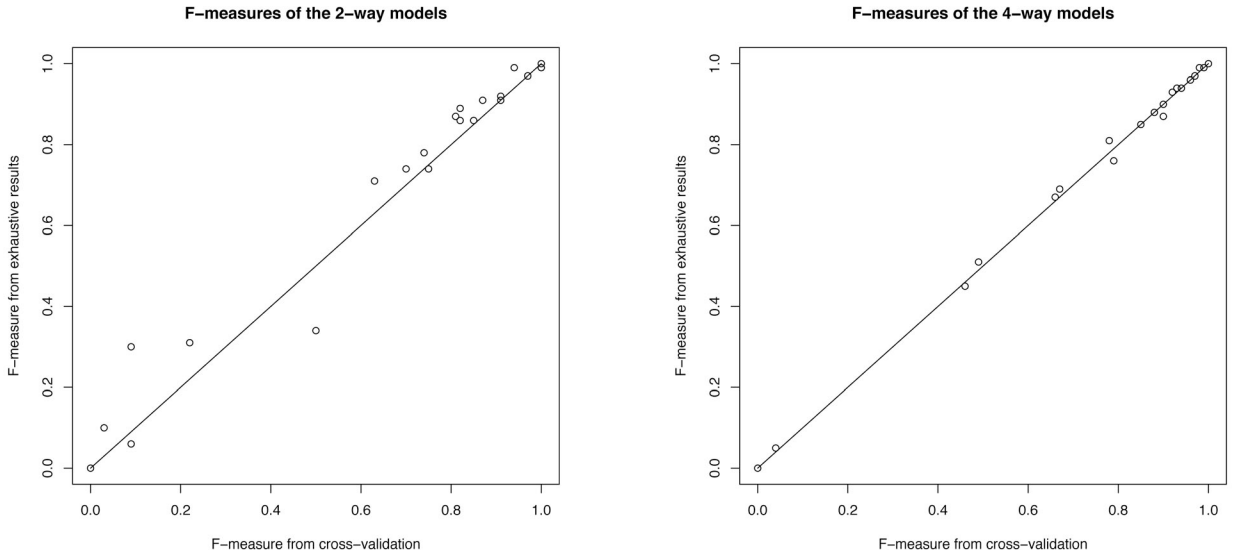


Fig. 10. Scatter plots of F measure for two-way and three-way models.

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 users of fixed-strength 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. 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).

The users of the variable-strength covering arrays, in addition to the guidelines above, need to know how to vary t across the entire configuration space. As we described in Section 2.2, VSCAs are desirable when it is too expensive to use a higher t for all options. Based on our experience, we present the following guidelines:

1. Leverage a priori knowledge of the system under test, if it is available. Information that leads to high-risk subspaces is valuable, e.g., information recommending that a subset of option combinations is more likely to cause failures, or that recent changes in the code base affect certain set of option interactions, etc. Consider assigning higher-level strengths to high-risk subspaces.
2. Leverage fixed-strength covering arrays to pinpoint high-risk subspaces, if no or limited reliable a priori information is available. Start with a fixed-strength covering array and analyze the resulting fault-characterization models to identify subsets of options that are highly correlated with the manifestation of failures. Consider assigning higher-level strengths to these subsets.
3. Leverage the fact that there may be some configuration options that dictate the size of the covering arrays. These options are the ones which have the largest number of settings. For example, in our experiments, the overriding option in the size of the covering arrays was the one static option with 29 settings. Consider manipulating the configuration space of nonoverriding options independently by assigning higher strengths. Depending on the configuration space, this strategy may provide higher strength coverage at no or reasonable cost. (See Section 5.1 for more details.)

7 Comparison with Random Schedules

In this section, we compare the effectiveness of t-way and randomly selected schedules. For this, we create 100 random schedules for each value of t where the size of each random schedule is the same as the corresponding t-way schedule. Since the CAs and VSCAs we create in this research are comparable in size, the results obtained from this section are also applicable to VSCAs unless otherwise stated. Our first concern is to see how well the random schedules reveal failures. Fig. 11 contains box plots 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 schedules tend to reveal slightly more failures than the corresponding random schedules with less variance.

Next, we evaluate the two approaches in terms of their fault characterizations. For this, we randomly choose 15 schedules for each value of t and create the classification-tree models for option-related failures. In general, we observe that random and t-way schedules yield comparable fault-characterization models.

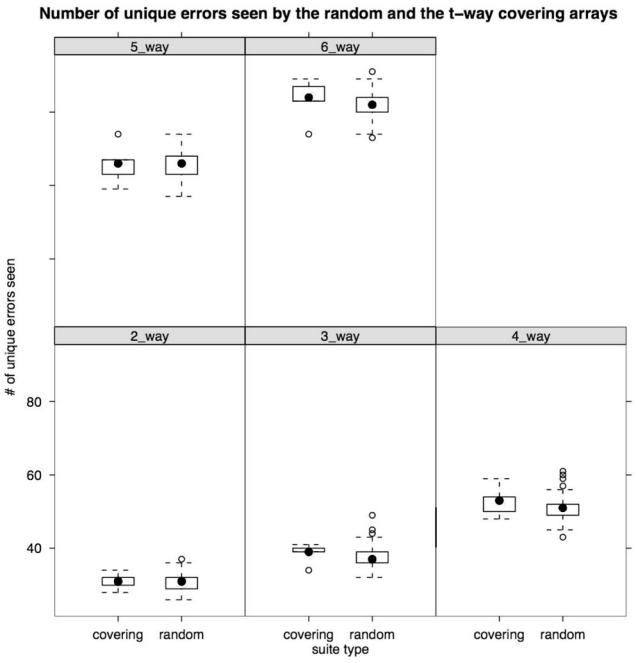


Fig. 11. Number of unique errors seen in random and t-way covering arrays.

Random schedules, however, sometimes completely miss option-related failures or result in unbalanced sampling of the failing subspaces. In the first situation, the models ignore the failure because it has 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 causes spurious options to be included in the models.

Fig. 12 illustrates this situation by contrasting the fault characterizations for test #2, error #18 obtained from the exhaustive schedule, a two-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 Fig. 12a. Compare this to the two-way schedule appearing in Fig. 12b. 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. The model created from the random schedule (Fig. 12c), however, 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 happens because, with the random schedule, when POLLER == 0, 86 percent of the configurations 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 two-way schedules, only 21 percent and 33 percent, respectively, of the configurations with ConnectionStrategy == 1 fail. This difference is simply due to an "unlucky" random selection that produced an unbalanced sampling of the underlying configuration space.

In summary, we observe that random and t-way schedules give comparable fault characterizations on average, but that the random schedules sometimes create unreliable models. Moreover, in practice, the covering-array approach automatically determines the size of the schedule, whereas there is no way to predetermine the correct size of a randomly selected schedule.

8 Related Work

Covering arrays frequently have been used to reduce the number of input combinations when testing a program [2], [3], [5], [8], [9], [12]. Mandl [12] first used orthogonal arrays, a special type of covering array in which all t-sets occur *exactly* once, to test enumerated types in ADA compiler software. This idea was extended by Brownlie et al. [2], who developed the orthogonal-array testing system (OATS). Their empirical results suggest that orthogonal arrays are effective in fault detection and provide good code coverage. Dalal et al. [8] argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults. In further work, Burr and Young [3], Dunietz et al. [9], and Kuhn and Reilly [10] provide more empirical results to show that this type of test coverage is effective. These studies focus on finding unknown faults in already-tested systems and equate covering arrays with code-coverage metrics [5], [9]. 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.

A structure similar to the variable-strength covering array is first suggested in [5] and termed "hierarchical test suites," but no empirical evaluation is provided. In [6], [7], Cohen et al. present a discussion, providing scenarios where variable-strength arrays might be useful. They develop a model to define VSCAs and present a construction technique. However, they have not applied these in practice. We do not know of any studies to date that

```
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)

Fig. 12. Fault characterization for test #2, ERR #18 obtained from the exhaustive schedule, a two-way schedule, and a random schedule, respectively.

provide empirical results comparing variable-strength arrays with their fixed-level counterparts.

Other techniques have been used to isolate faults in code for debugging [11], [20]. The bug isolation project uses code instrumentation and statistical sampling to achieve fault localization [11], while the delta debugging project isolates minimal subsets of tests that cause faults through successive elimination of the input space [20].

Our research is unique because it uses a two-dimensional approach. First, we statically reduce the configuration space that will be tested for cost efficiency (i.e., we decide which configurations to test). After testing, we analyze the results with a classification algorithm for effective fault characterization. Although delta debugging also decides which subset to test, this is done dynamically; therefore, the cost of testing is unknown at the start.

9 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 first recast the problem of selecting test schedules (determining which configurations to test) as a problem of calculating a fixed-strength, 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. In our initial study [19], we examined the results obtained using only one operating system (Linux). In this study, we replicated the experiments on a second operating system (Windows). Although the individual results for each operating system were slightly different, we were able to draw the same conclusions.

- 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 percent 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 showed that we can improve the fault-characterization accuracy at a low construction cost by combining lower-strength covering arrays rather than increasing the covering- array strength.

We were also able to develop diagnostic tools to support software practitioners who want to use fixed-strength 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 nonoption-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.

To further improve the fault-characterization process, we extended our work from [19] to test the effects of using a different kind of covering array called a variable-strength covering array as a sampling strategy. Variable-strength covering arrays, unlike their fixed-strength counterparts, allow us to test higher-level interactions only in subspaces where they are needed (i.e., in high-risk subspaces), while keeping a low level of coverage across the entire space. We developed several models of variable-strength arrays to focus testing on the runtime options. The sample sizes were close to those of the fixed-strength arrays, allowing us to make comparisons. To gain a better insight into the usefulness of these arrays, we conducted a simulation that seeded four-way interaction faults into our configuration space. We observed that variable-strength arrays slightly improved the efficiency of the fault-localization process in two ways:

- They reduced the cost of the process without compromising its accuracy.
- For the same cost, they improved the accuracy of the process.

We also provided users of variable-strength covering arrays with guidelines on how to vary t across the configuration space.

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. One potential threat is the representativeness of the ACE+TAO subject applications, which, though large, are still just one suite of software systems. A related issue is that we have focused on a relatively simple and small subset of the entire configuration space of ACE+TAO; the actual configuration space is much larger. While these issues pose no theoretical problems, we need to apply our approach to larger, more-realistic configuration spaces in future work to understand how well it scales.

In continuing work, we are integrating covering-array 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 interoption constraints on the fault characterizations.

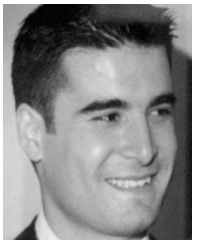
As future work, we plan to make the current fault-characterization process an iterative and adaptive process. The idea is to start with a low-strength covering array, analyze the results on the fly as they are returned to identify *hot spots* in the configuration space, and then test higher-level interactions only in these hot spots (e.g., via variable-strength covering arrays) by sequentially adding new configurations to the current testing schedule. Such a process can be especially useful when we need faster feedback and when we need adaptation based on resource availability.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This material is based on work supported by the US National Science Foundation under an NSF EPSCoR First Award, by grant ITR CCR-0205265, and by US Office of Naval Research grant N00014-05-1-0421.

REFERENCES

[1] L. Breiman, J. Freidman, R. Olshen, and C. Stone, *Classification and Regression Trees.* Wadsworth, 1984.
[2] R. Brownlie, J. Prowse, and M.S. Phadke, "Robust Testing of AT&T PMX/StarMAIL Using OATS," *AT&T Technical J.,* vol. 71, no. 3, pp. 41-47, 1992.
[3] K. Burr and W. Young, "Combinatorial Test Techniques: Table-Based Automation, Test Generation and Code Coverage," *Proc. Int'l Conf. Software Testing, Analysis, and Review,* 1998.
[4] M. Chateauneuf and D. Kreher, "On the State of Strength-Three Covering Arrays," *J. Combinatorial Designs,* vol. 10, no. 4, pp. 217-238, 2002.
[5] 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 Trans. Software Eng.,* vol. 23, no. 7, pp. 437-444, 1997.
[6] M.B. Cohen, C.J. Colbourn, J. Collofello, P.B. Gibbons, and W.B. Mugridge, "Variable Strength Interaction Testing of Components," *Proc. Int'l Computer Software and Applications Conf. (COMPSAC),* pp. 413-418, 2003.
[7] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge, "Constructing Test Suites for Interaction Testing," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 38-44 2003.
[8] 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," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 285-294, 1999.
[9] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 205-215, 1997.
[10] D. Kuhn and M. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," *Proc. 27th Ann. NASA Goddard/IEEE Software Eng. Workshop,* pp. 91-95, 2002.
[11] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 141-154, 2003.
[12] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Comm. ACM,* vol. 28, no. 10, pp. 1054-1058, 1985.
[13] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D.C. Schmidt, and B. Natarajan, "Skoll: Distributed Continuous Quality Assurance," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 459-468, 2004.
[14] C.V. Rijsbergen, *Information Retrieval.* London, UK: Butterworths, 1979.
[15] D. Schmidt, D. Levine, and S. Mungee, "The Design and Performance of the TAO Real-Time Object Request Broker," *Computer Comm.,* special issue on building quality of service into distributed systems, vol. 21, no. 4, 1998.
[16] D.C. Schmidt and S.D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns.* Boston: Addison-Wesley, 2002.
[17] I.H. Witten and E. Frank,, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.
[18] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. Schmidt, A. Gokhale, and B. Natarajan, "Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 293-302, 2005.
[19] C. Yilmaz, M.B. Cohen, and A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA),* pp. 45-54, 2004.
[20] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 183-200, 2002.



Cemal Yilmaz received the BS and MS degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1997 and 1999, respectively. In 2002 and 2005, he received the MS and PhD degrees in computer science from the University of Maryland at College Park. He is a postdoctoral researcher at the IBM Thomas J. Watson Research Center, Hawthorne, New York, where he works in the field of software quality assurance. His research interests include distributed, adaptive, and continuous quality assurance, applications of formal methods to software testing, fault localization, software performance modeling, evaluation, and optimization, and highly configurable software systems.

Myra B. Cohen received the BS degree from the School of Agriculture and Life Sciences at Cornell University, the MS degree in computer science from the Univesity of Vermont, and the PhD degree in computer science from the University of Auckland, New Zealand. She is an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska–Lincoln. She is a member of the Laboratory for Empirically-Based Software Quality Research and Development (ESQuaReD). Her research interests include software interaction testing, testing of configurable systems, metaheuristic search, and applications of combinatorial designs. She is a member of the IEEE.

Adam A. Porter received the BS degree summa cum laude in computer science from the California State University at Dominguez Hills, Carson, Calif., in 1986. In 1988 and 1991, he received his MS and PhD degrees, respectively, from the University of California at Irvine. Currently an associate professor, he has been with the department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland since 1991. He is a winner of the National Science Foundation Faculty Early Career Development Award and the Dean's Award for Teaching Excellence in the College of Computer, Mathematics, and Physical Sciences. His current research interests include empirical methods for identifying and eliminating bottlenecks in industrial development processes, experimental evaluation of fundamental software engineering hypotheses, and development of tools that demonstrably improve the software development process. He is a senior member of the IEEE.