

Incremental Covering Array Failure Characterization in Large Configuration Spaces

Sandro Fouché
Department of Computer Science
University of Maryland
sandro@cs.umd.edu

Myra B. Cohen
Department of Computer Science
and Engineering
University of Nebraska-Lincoln
myra@cse.unl.edu

Adam Porter
Department of Computer Science
University of Maryland
aporter@cs.umd.edu

ABSTRACT

The increasing complexity of configurable software systems has created a need for more intelligent sampling mechanisms to detect and characterize failure-inducing dependencies between configurations. Prior work – in idealized environments – has shown that test schedules based on a mathematical object, called a covering array, in combination with classification techniques, can meet this need. Applying this approach in practice, however, is tricky because testing time and resource availability are unpredictable, and because failure characteristics can change from release to release. With current approaches developers must set a key covering array parameter (its strength) based on estimated release times and failure characterizations. This will influence the outcome of their results.

In this paper we propose a new approach that incrementally builds covering array schedules. This approach begins at a low strength, and then iteratively increases strength as resources allow. At each stage previously tested configurations are reused, thus avoiding duplication of work. With the incremental approach developers need never commit to a specific covering array strength. Instead, by using progressively stronger covering array schedules, failures due to few configuration dependencies can be found and classified as soon and as cheaply as possible. Additionally, it eliminates the risks of committing to overly strong test schedules.

We evaluate this new approach through a case study on three consecutive releases of MySQL, an open source database. Our results suggest that our approach is as good or better than previous approaches, costing less in most cases, and allowing greater flexibility in environments with unpredictable development constraints.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSA '09, July 19–23, 2009, Chicago, Illinois, USA.

Copyright 2009 ACM 978-1-60558-338-9/09/07 ...\$5.00.

1. INTRODUCTION

As software systems grow in complexity, so too grows the difficulty of testing them. Systems are no longer deployed as a single program, but as entire ecosystems of inter-dependent software entities. Each of these entities brings with it individual features, flaws, performance profiles and configuration parameters, all of which together impact the overall system behavior. Understanding this web of inter-dependencies rests, in part, on effective software testing since these dependencies are often uncovered through the discovery of subtle *interaction* failures; failures due to specific combinations of features. Ironically, the same complexity that makes testing so important, leads to a combinatorial explosion of system configurations [5, 16]. Exhaustively testing every possible configuration, is difficult, if not impossible. Consider for instance, MySQL [13] the experimental subject of this paper; which has over 72,000,000 configurations in just the small slice of its configuration space we examine in this paper. The full configuration space is much larger. Confounding this issue is the fact that modern software systems often evolve rapidly and that the amount of time and hardware resources available to the testing process are often constrained and time-varying [9]. Therefore, testing processes must be fast and efficient if the information they provide is to be useful.

Historically, techniques for testing systems with large configuration spaces include: testing just (a potentially small set of) default configurations, randomly choosing some subset of configurations for testing, or choosing test configurations based on developer intuition and experience. While all of these methods can find flaws in a software system, they have several drawbacks. First, they cover a limited, perhaps biased, sub-space of configurations. Furthermore, unless the test configurations are updated over time, testing will continue in already debugged configurations while leaving large sections of the configuration space unexplored. To improve this, several researchers have studied configuration sampling strategies based on computing mathematical objects called covering arrays [1, 6, 10]. This approach generates a test schedule that satisfies specific coverage metrics, that of testing all t -way combinations of the configuration options.

In previous work, Yilmaz et al. [16] extended this approach and integrated it into the Skoll system [12], which is a distributed continuous quality assurance (DCQA) environment that allows for highly parallel execution of QA processes. In that work covering arrays were used to generate test schedules. Those schedules were then executed in parallel across a grid of computers, the results were returned to

central servers where uncovered failures were automatically classified to help developers find their underlying causes. Their results suggested that the covering array test schedules produced better classification models than equivalently-sized random samples and that the process scaled reasonably well to large configuration spaces.

This extension, while promising, still has some limitations. First, covering arrays depend on developer insight to select the key sampling parameters. In order to reliably classify failures that are caused by t configuration options, samples must, at a minimum, test all t -way combinations of these options. This means the tester must know *a priori* what *strength*—size of t —to use. If they set t too large, resources will be wasted or the process may not complete before the next release; while selecting t to be too small, may result in poor classification and require repeating the process with a higher strength for t . Since systems often have multiple failures with different causes, either—or even both—of these situations is virtually guaranteed.

Second, developers must choose how many covering arrays to generate at each level of t . A t -way covering array, by design, computes a sparse sample of a configuration space. These samples have the property that they can *detect* interactions between any t options. They cannot be relied on, however, to correctly *characterize* which specific options and settings are actually interacting (because of ambiguities in the sparse data). As a result classification techniques can mistake coincidental relationships for actual failure causes. In addition, problems involving more than t options can be incorrectly classified; as can non-deterministic failures, which, in turn, misdirects debugging attempts, wasting developer time and effort. Although not widely discussed in the research literature, a common way to limit these problems is to run multiple different covering arrays, hoping that the differences in the specific configurations tested can disambiguate actual from accidental relationships.

Third, because it is not generally possible to use an arbitrary initial portion of a t -way covering array to reliably classify failures caused by fewer than t options, developers should ideally run the covering array as a unit, waiting until all tests have been run before classification can start. In this situation, there is no way to ensure that failures are found and classified as early as possible. All told this approach can run more tests than necessary, mis-correlate failures with configuration parameters, duplicate work, and suffer delays in reporting classification information.

This paper addresses these limitations by redesigning the underlying covering array technique. An early prototype of this approach was described in [8]. This paper extends that initial effort with improved algorithms, weakened assumptions, and much more thorough empirical evaluation. The redesigned approach is *incremental*. It begins with low strength covering arrays and failure classification to provide early results to developers. It then uses incrementally stronger covering arrays as results and resources indicate. A central tactic of this approach is to lower the cost of incremental execution by carefully reusing results from earlier test runs. It thereby efficiently increases covering array strength as far as time and resources allow. We call the new approach: *incremental covering array failure characterization*. We have applied our incremental approach across a large configuration space ($\sim 72\text{M}$ configurations) of the open source data base, MySQL [13], and evaluated it across three

Number	Config			Result
	o1	o2	o3	
1. *	0	0	0	FAIL
2.	0	0	1	FAIL
3.	0	0	2	FAIL
4.	0	1	0	PASS
5. *	0	1	1	PASS
6. *	0	1	2	PASS
7.	1	0	0	PASS
8. *	1	0	1	PASS
9. *	1	0	2	PASS
10.*	1	1	0	PASS
11.	1	1	1	PASS
12.	1	1	2	PASS

Table 1: An exhaustive schedule

consecutive releases, each with varying characteristics. For this data, we then show that, without compromising effectiveness, our technique is in the worst—but quite unlikely—case less than 5% more expensive than the traditional approach, while, in other more likely cases, it reduces effort by as much as 27%. We also show that this technique can produce accurate and actionable debugging information far earlier than traditional techniques.

The remainder of this paper is organized as follows: Section 2 briefly reviews the mathematical tools and process we used; Section 3 provides a field study of about 50 MySQL releases that motivates our process. Section 4 describes the revised covering array algorithm and the incremental strategy; Section 5 describes our empirical evaluations; Section 6 compares incremental covering arrays to other scheduling policies; and Section 7 presents concluding remarks and possible directions for future work.

2. BACKGROUND

In this section we provide some background on our techniques. We first explain how failure characterization works and then describe the basis for determining which configurations to test: covering arrays. Finally we present Skoll, the distributed quality assurance environment in which we execute our large scale test processes.

2.1 Failure Characterization

Failure characterization aims to provide developers with compact and accurate descriptions of failing configuration subspaces. This section details both the process and how we evaluate its performance. Table 1 depicts the results of exhaustively testing a system with three configuration options ($o1$, $o2$, and $o3$), where the first two options are binary (0, 1) and the third is ternary (0,1,2). There are no constraints among the options, so there are 12 valid configurations. Each configuration has an outcome indicating whether it PASSEd or FAILed. To automatically determine the likely cause of the three failures we feed the data to a classification tree analysis technique. This produces the classification tree shown in Figure 1 and indicates that the failures appear whenever $o1 == 0$ and $o2 == 0$.

Classification trees use a recursive partitioning approach to build a tree-structured model [15] that correlates a configuration’s test result (e.g., passing or failing) with the settings of its options. Thus, each node denotes an option, each edge represents an option setting, and each leaf represents a test outcome or set of outcomes (if there are, for example, different failure types). In this work we use covering array test

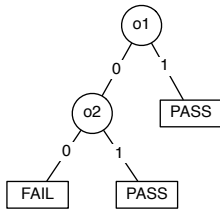


Figure 1: An example classification tree

schedules (described next) to generate data for classification and then use a more comprehensive data-set to evaluate the resulting classification models.

2.2 Covering Arrays

While the model in Figure 1 explains the failures in the underlying dataset, it does so at the cost of gathering exhaustive testing data. We can, however, obtain the same model using only a subset of the exhaustive data (the starred configurations in Table 1). This reduced schedule is only one-half of the size of the exhaustive one. We selected these configurations because they constitute a 2-way *covering array* (CA) [1, 2] of the configuration space; i.e., all pair-wise combinations of configuration option settings appear in at least one of the starred configurations. t -way covering arrays “cover” all possible combinations of option settings between any t options in the configuration space. Importantly, they do this while also limiting the total number of configurations tested. The approach takes k configuration options (or *factors*) each with v_i settings (or *values*), and produces an $N \times k$ array over the values of v_i in the corresponding columns. The resulting covering array $CA(N; t, k, (v_1 v_2 \dots v_k))$ has N configurations with the property that every $N \times t$ sub-array contains all ordered subsets of size t from each of the v_i ’s at least once. The parameter t , the covering array *strength*, corresponds to the number of options whose combinations of option settings must be covered. For instance if we set $t = 2$ we would need to cover all combinations of option settings for every pair of options; if we set $t = 3$ we would need to cover all possible combinations of option settings for every trio of options. The starred configurations in Table 1 can be written as a $CA(6; 2, 3, (2, 2, 3))$ or more compactly as a $CA(6; 2, 2^2 3^1)$, which uses a shorthand notation, where the superscript represents the number of times a particular v is repeated: We drop the k , since it is implicit in this notation.

Theoretically, covering arrays can have an arbitrarily large number of rows (configurations), N , but there are a number of heuristic algorithms for constructing minimally-sized covering arrays [1, 2, 14]. While most of these algorithms provide some way to construct a covering array on top of already seeded test cases (a requirement for our incremental approach) not all provide the ability to handle inter-option dependencies or *constraints*. The subject studied in our case study has some constraints that we must consider, so we are limited to tools that incorporate this functionality. In recent work [4] Cohen et al. extended two algorithms for constructing covering arrays for this purpose. The first is a meta-heuristic search technique, simulated annealing [2]. The second is a greedy algorithm based on the automatic efficient test case generator (AETG) [1]. In general the simulated annealing algorithm produces smaller test suites, but as discussed in [4] the current constraint version does not yet scale well. It suffers from extremely long run times at

higher strengths. Consequently, in this paper we use simulated annealing when possible (for $t = 2$ and 3), and switch to the greedy algorithm for $t \geq 4$.

2.3 Skoll

Skoll [12] is a process and infrastructure that simplifies executing QA tasks across a grid of computing resources. It uses an intelligent steering agent (ISA) to plan and distribute the QA process. Skoll efficiently leverages computing resources by dividing global QA processes into multiple sub-tasks which can then be distributed to client machines and executed. The results, when returned, are fused together to complete the overall QA process. Skoll maintains a formal model of the QA processes’ configuration space that captures configuration options and their settings as well as constraints (refer to [12] for further details).

In this paper, we create covering arrays for a configuration model and use Skoll to distribute and test individual configurations. For a given configuration, the client configures and compiles the source code, runs tests, and sends the results back to the Skoll database. We then merge the results and build classification trees to characterize failures.

3. FIELD STUDY

We conducted a field study using Skoll to run a continuous build, integration and test (CBIT) process for the MySQL database project based on covering array test schedules. We applied the traditional approach from [16] to process a partial configuration space of MySQL, with over 72,000,000 unique configurations. Given the large size of the configuration space, exhaustive testing was not possible. Exhaustively testing even this limited configuration space would require approximately 217,645,056 machine-hours! Thus, we opted to use a covering array approach. During this study, we uncovered several limitations of the *traditional* covering array test schedule technique, motivating the need for our incremental approach. In this section we describe our experiences and highlight each of the limitations we faced.

3.1 Our MySQL CBIT Process

MySQL is an open-source, multi-threaded, SQL database management system (DBMS) [13]. Initially released 12 years ago, its various components contain 2+ million lines of code. It has been downloaded 10+ million times and is available for use on over 20 platforms. MySQL has a significant number of test cases (including both installation tests and generic SQL tests), and MySQL enjoys a large developer community that actively updates and tests the system.

Configuration Model. MySQL runs on a myriad of hardware architectures and operating system combinations and allows extensive customization of functionality. This is supported through the use of configuration options. For example, MySQL can be compiled with support for differing character sets and back-end storage engines. In this paper we consider only partial, but still large, subset of MySQL’s configuration options.

Continuous Build, Integration and Test. Our Skoll-based MySQL CBIT process builds and integrates a MySQL instance and then executes a standard battery of 772 tests against the resulting executables. Multiple configurations are tested in parallel using a grid of computing resources primarily located at the University of Maryland. Because our approach shares CBIT effort across a large grid, many

more system configurations can be tested than is possible with MySQL's limited in-house testing resources.

3.2 Problems Encountered

Our initial implementation of this process used a traditional covering array test schedule to define the configurations to be tested whenever new code was checked into the main developer repository. With this method we first determined the desired strength of the covering array test schedule (i.e., $t=2,3$, etc.) and then ran the schedule generated for this strength. Once the covering array test schedule was finished we classified the failures. Applying this approach to the live MySQL development process however, we very quickly ran into problems. Specifically, the standard covering array approach required us to make several problematic decisions/assumptions. Each such decision forced us to statically fix something that varied in reality; and each such mismatch implied unnecessary costs and/or reductions in effectiveness. We describe these in detail below.

Variability of Test Time and Resources. The time needed to test one MySQL version in one configuration varies depending on the features and failure rates of the configuration. Successful test runs on limited feature sets (which skip some inapplicable tests) complete in as little as 45 minutes. Other test runs, exercising larger feature sets and possibly failing on several timeout values, take upwards of 4 hours to execute, with the average being about 3 hours across all configurations. Thus, for a given amount of testing time, it was impossible to know how many configurations might be tested. This hindered our ability to choose a strength of t that would fit into a given period of time.

Second, and more importantly, the time available to test a given system version varies considerably. During a six month period, we saw versions that were current for a few minutes; some for a few days and others for more than a month. In fact, the total time available for testing a version is not known until the *next* source change is committed. As a result, we often selected the wrong value of t and committed to test schedules that were far too large or too small for a particular release period.

Third, because some of our resources are volunteered or shared, the number and capacities of test resources could not be known a priori. Again our test planning was affected because we never knew how many CPUs would be available for testing and for how long we could use them.

Variability Caused by Non-Deterministic Failures. During testing we uncovered several non-deterministic failures; i.e., they failed during some test runs but not others. As we discussed in [16] this can cause misleading classifications and wasted developer effort. This is problematic for the traditional process, because it only runs a single covering array test schedule at the chosen strength.

Variability of Failure Characteristics. Finally, the characteristics of failures can change from version to version. For instance, in one random sample involving 14 versions checked in over about 2-3 weeks, we saw a variety of failures whose root causes involved anywhere from 1 to 5 configuration options. In many cases, the patterns and their distributions changed across releases. For example, in one release we observed failures caused by 1 to 4 options, distributed respectively as: 24%, 38%, 34%, and 2% of the total failures. In another release we documented failures caused by 1,2,3 or 5 options, distributed respectively as follows: 5%, 48%,

45%, 0% and 2%. Therefore, it was once again impossible for us to choose, a priori, the optimal covering array strength for testing.

3.3 Further Problem Analysis

Traditional covering arrays force developers to commit to a particular test schedule without knowing whether it has the right sampling characteristics and without knowing whether they can finish it before the next version arrives. And as mentioned earlier in Section 1, choosing incorrectly can have severe consequences. For example, when a version has simple failures but large test schedules are used, then testing will require much more work than necessary. If the test schedule, however, is too large to finish before the next version arrives, then failure characterization performance may be negatively affected. Also, if failures are complex, but the test schedule is too small, failure characterization will suffer and the process will need to be repeated at a higher strength.

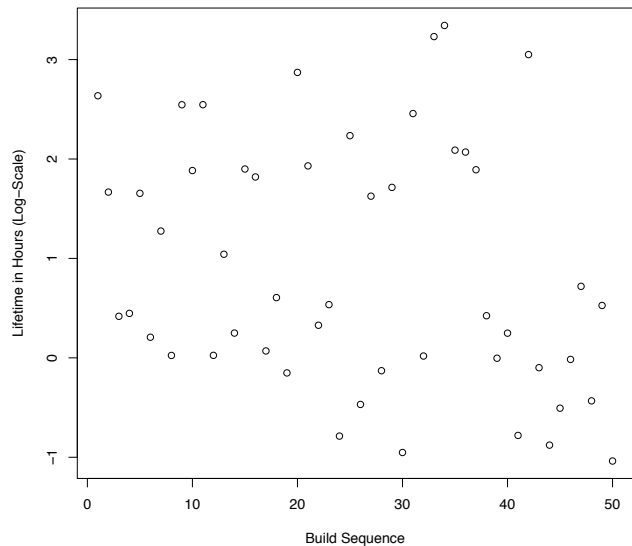


Figure 2: Lifetime of 50 releases of MySQL

Since the update frequency for specific MySQL versions can vary (sometimes multiple times a day, sometimes every few days), testing results have varying useful lifetimes. The variance in the lifetime of MySQL releases was a major motivating factor for our approach. We examined historical build data from the MySQL database project. Figure 2 shows a graph of the lifetime in hours (on a \log_{10} scale) for 50 consecutive builds between March 2007 and January 2008. We can see a range of lifetimes ranging from less than one hour to a maximum of about 2,000 hours. Clearly the ability to test such systems and our failure characterization techniques will depend on this variance. The differences in lifetimes have multiple causes. Sometimes geographically-distributed developers happen to check in code at nearly identical times. Sometimes a check-in is so obviously flawed that it is pulled quickly. Sometimes the mix of development activities can vary. That is sometimes developers are more focused on cleaning up existing bugs, while at other times they are more focused on adding new features (which tend to arrive at a slower pace). It is clear that we cannot always predict ahead of time, what time frames are available for testing, or what the complexity of failures will be.

The next analysis we performed was to understand what would happen if we always chose to run the lowest strength covering array test schedule first. For instance, if we always select $t = 2$, then we can increase our test strength if additional time is left. In this approach it might seem that we can reuse some of the already tested t -way test schedule. But in fact, if we use the traditional approach and build a new $t + 1$ covering array test schedule there may be little or no overlap between the old and the new test schedules. In fact, for our system we would have had to re-do almost all of the work from the prior test schedule. Most of the algorithms for building covering array test schedules make some random decisions in the array construction, meaning that each time the algorithm is run a different array is generated. Even in algorithms that use determinism there is no guarantee that a $t+1$ -way array will have any resemblance to a t -way array. Although, one can use techniques presented – called *seeding* – to build upon lower strength arrays (and this will be leveraged in our new approach) this is not the ordinary construction mode for covering arrays.

To test our hypothesis that different strength covering arrays have almost no overlap in tested configurations, we generated 10 arrays for MySQL at each strength of $2 \leq t \leq 5$. We then compared each of the t -way arrays against all of the $t + 1$ -way arrays providing comparisons between consecutive strengths. This gave us 100 data points for each increment of t . In the analysis we computed the number of configurations shared between the t -way and each of the 10 $t + 1$ -arrays. We found that there were zero configurations in common between any of the 10 t and $t + 1$ arrays in this data set. Given that there are over 72,000,000 possible configurations this is not too surprising. It highlights the fact, that choosing too low an initial strength causes redundant work and wasted effort.

Finally, we tried to determine what would happen if we simply chose the highest strength t we were likely to need and simply ran them *as far as possible* to completion. In this scenario, we would be unlikely to finish many of our test schedules. As a result characterization of t -way failures may be impossible because the data set is incomplete. However, since a t -way covering array is also a $t - 1$ -way, for $t > 1$, we might still get reasonable characterization performance for some failures. To better understand this issue, we examined the 10, 3 and 4 way arrays to see how quickly $t - 1$ -way combinations are covered when running a t -way covering array. We found that we generally needed to run substantially more configurations of the t -way array to achieve all $t - 1$ -way coverage than we would have needed by just running the $t - 1$ -way array initially.

Although the coverage grows quickly there is a long plateau before the full coverage is achieved. For instance with the 3-way arrays we needed to test 58 configurations on the average, to achieve 2-way coverage (versus 22 configurations for an average 2-way covering array). Running the 4-way arrays, we needed an average of 46.2 configurations to complete 100 percent of the 2-way coverage while we needed an average of 174.2 configurations to reach 100 percent of the 3-way coverage (versus 80 for an average 3-way covering array). The main implication, is that if we are unable to complete a large portion of a given test schedule, then we may not be able to classify even the lower strength failures because our information sample will be incomplete.

4. PROPOSED SOLUTION

From our perspective, the three major limitations of traditional covering array schedules for failure characterization are (1) the lack of guidance for selecting an initial interaction level (2) the inability to reuse information from prior test runs if the initial interaction level proves too low and (3) that non-deterministic failures can be indistinguishable from higher-level interaction failures. To address this problem we have designed and evaluated an incremental process for creating and using covering array test schedules. With this new approach, we begin by testing at the lowest strength (i.e., $t=2$), and then successively move to higher strengths. At each stage, we construct the next higher-strength covering array so that it incorporates, to the largest extent possible, the configurations already run in lower-strength covering arrays. Thus, we only need to run the subset of new configurations to cover the current strength. We have designed two variants of this process: one that creates a single covering array at each strength and one that creates multiple covering arrays at each strength (for handling non-deterministic failures). Developers must choose which variant to apply to any given system. This process allows classification as early as possible, and improves testing efficiency. It also allows us to generate test schedules in environments with unknown time and hardware resources.

Our key technical conjectures are that we can construct each covering array using a *seed* taken from already run lower strength arrays and that their size will be approximately that of a traditionally built covering array. Seeding means that we *fix* a set of configurations at the start, and construct the new covering array by filling in the required t -way interactions not already contained in the seed.

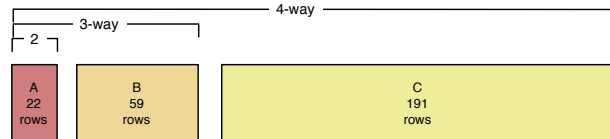


Figure 3: Seeding to create a covering array

4.1 A Single Array at Each Strength

Figure 3 illustrates this approach in its simplest form. Using the model in Table 2 (where $k = 23, v_1..v_k = 2^{18}3^34^15^1$), we start with an initial 2-way array called A, created using traditional techniques. To later create a 3-way array, A+B, we will use A as a seed and fill in the remaining rows, B. Similarly, we can build a 4-way covering array A+B+C using A+B as a seed.

If our incremental covering arrays are roughly the same size as traditional ones, then the 2-way array would have about 22 configurations and the 3-way, 80. We would execute the 2-way array first and then attempt to classify any failures involving two options. Because we build the 3-way covering array using the previous 2-way array as a seed, the 22, 2-way configurations are reused and only about 58 new ones would have to be run to get complete 3-way coverage. We would be able to classify all 2-way failures after only 22 configurations and we could still classify all 3-ways after running only a total of 80 configurations, i.e., it costs us no more than before to execute the 3-way array, but we get early classification of the 2-way failures.

A key unanswered question is whether the covering ar-

rays built with seeds will be comparable in size to the traditionally built ones. Although seeding has been used in construction of covering arrays, seeds have been small, used mostly to provide a way to ensure coverage of a *default* set of configurations [1]. It has also been used as a construction technique, for the purpose, of generating small covering arrays, that in some cases represent previously unknown upper bounds [3]. Little is currently known, however, about the size of covering arrays generated by large seeds. We have found heuristically that we can use a seed of approximately the same size as a $t - 1$ array to build a t -way array for our problems. Beyond this our solutions degrade. We use this heuristic in our case study and leave a generalization for future work.

4.2 Multiple Arrays at Each Strength

As mentioned earlier, a single covering array does not always provide enough information to definitively associate a failure with any particular t -way configuration option interaction. This is because the failure might be caused by: (1) an actual t -way interaction, (2) a higher-order interaction that is coincidentally being uncovered, or (3) a non-deterministic failure of the system that merely happens to occur on this execution. For example, a given 2-way covering array might indicate that activating option X and option Y simultaneously leads to a failure, but in reality the presence of options X, Y, T , and R might be required to trigger the failure, and this run just happens to exercise that configuration. Similarly, two covering arrays of any given strength may not be sufficient to disambiguate a random, non-deterministic failure from a higher-order failure.

To deal with this situation, testers in practice may elect to run more than a single covering array test schedule at each strength, t . Therefore, we have extended our incremental approach to gather and distribute already tested configurations across multiple seeds when building multiple covering arrays at higher strengths. A prototype of this idea appears in earlier work [8], but that algorithm was ad hoc and our evaluation used a simplified model of the MySQL software subject to calculate the potential cost savings if an incremental approach would have been followed. We formalize the procedure in the following algorithm, and have refined the configuration model to contain more complex inter-option dependencies required in actual testing. We then use this model and our incremental algorithm for failure characterization in three different releases of a real configurable software system in our case study.

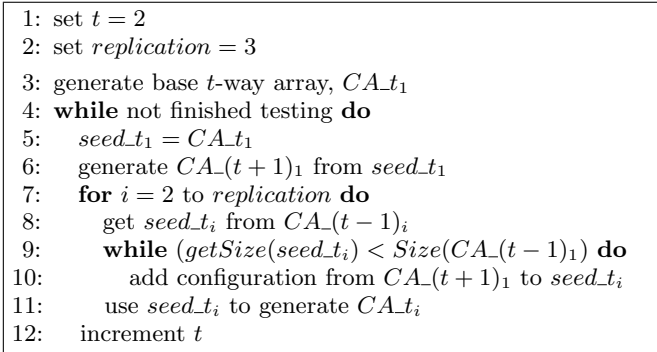
4.3 Incremental Covering Array Algorithm

The pseudocode for our algorithm is shown in Algorithm 1. We begin by allowing users to specify both the starting strength (t) and the number of arrays to generate at each strength (replication) (Lines 1 and 2). In this example, we fix these at 2 and 3 respectively. Next we generate the first t -way array using the traditional approach (Line 3) and begin constructing arrays iteratively at higher strengths until we are finished testing (Lines 4-12). Each iteration starts by using the first t way array, CA_{t_1} as a seed to build the first $t+1$ array, $CA_{(t+1)_1}$ (Lines 5-6). For each of the remaining arrays at strength t , $CA_{t_2}..CA_{t_i}$ (for loop: Lines 7-11), we gather all non-seeded configurations from $CA_{(t-1)_i}$ (Line 8) and additional configurations from $CA_{(t+1)_1}$ (Line 9-10) to create the seed for CA_{t_i} . The seed should match

the size of $CA_{(t-1)}$. We then build CA_{t_i} using this seed (Line 11). Finally, we increment t (Line 12) and repeat the outer loop.

Figure 4 depicts an example of this strategy that starts at $t = 2$ and creates three arrays at each strength. The matching letters and arrows show seeding relationships. For instance, the 2-way array, A, becomes a seed for the first 3-way array and the configurations from the first 3-way array labeled as B and C are used as seeds for the second and third 2-way arrays. We can see that second and third 3-way arrays have seeded configurations (E,G,F,H) that are taken from both the 2-way arrays (E,F) and from the first 4-way array (G,H) - not shown. In the first iteration, when $t = 2$ we see a special case. Since there is no $t - 1$ -way array, the seed set is initially empty (Line 8); in this case, all of the seeded configurations come from the higher strength array (B and C).

In essence, our algorithm iterates through a process of building a covering array of strength $t + 1$, using a strength t array as a seed; it then uses configurations from both previously run $t - 1$ -way arrays and from the newly created $t + 1$ array to seed the building of the remaining t -way arrays. Thus, after running some or all of the t -way covering arrays, a large portion of the initial $t + 1$ -way array has already been run. At this point, t -way failures can be classified and developers can begin to fix the underlying faults. Meanwhile, if developers wish to, they can increment t by 1 and repeat the process. Now, a new $t + 1$ -way array will be constructed using the first t -way array as a seed. Part of that $t + 1$ -way array is combined with the configurations for the $t - 1$ -way arrays and used as seeds to generate the rest of the t -way arrays.



Algorithm 1: INCREMENTAL

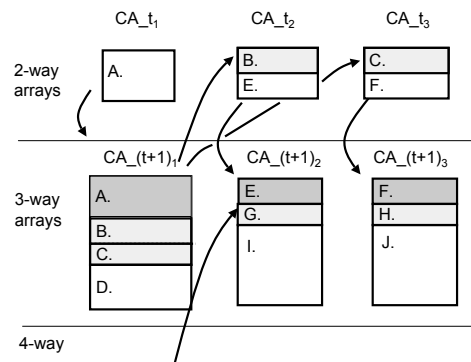


Figure 4: Constructing iterative CAs

5. A CASE STUDY

To determine the feasibility of applying the incremental approach to failure characterization we applied our technique to three consecutive releases of MySQL (build 1.2510 – 1.2512). We also applied the traditional covering array approach at varying strengths of t . Our goal is to compare the costs and benefits of the modified approach to those of the traditional covering array approach, which requires pre-selection of the covering array’s strength, analysis of the resulting test data after all tests are completed, and, if necessary, repetition of the process with a higher strength covering array.

5.1 Methodology

Our subject program for these studies is the MySQL client and server system [13], specifically focusing on 3 later releases of MySQL 5.1 that were not part of the field study described earlier in Section 3. For this study our configuration space comprises 23 MySQL features (18 binary, 3 with 3 values, 1 with 4 values and 1 with 5 values) There are several constraints on these options, which give rise to 8 explicit pair-wise combinations that cannot occur in the model. The resulting configuration space contains 72,548,352 possible configurations. Table 2 gives more details of our configuration model and its constraints.

We tested these configurations using 50 nodes of our Skoll Quality Assurance Cluster. Each node runs Red Hat 3.4.4-2 on a 2.8 GHz Pentium 4 with 1GB of memory. During testing, the configuration model, individual test plans and all results were stored in the Skoll Cluster Database Server, running MySQL 5.0.27. For these 3 releases, we tested a total of 5994 compilable, valid configurations, running 772 developer-supplied regression tests on each. Each test was designed to emit an error message in the case of failure, and we captured and recorded all results. All told this testing took 2 machine years to complete and included over 4.6 million individual test executions. These results form the basis for our analysis in this paper.

Covering Arrays. Using the configuration model previously described, we executed both the traditional and incremental covering array algorithms. We created 3 incremental covering arrays for each value of t from 2 through 4, and a single incremental 5-way covering array (this was needed to generate the incremental 4-way arrays). Specifically we computed a $CA(N; t, 2^{18}3^34^{15}5^1)$ for each value of t . For comparison purposes, we generated three traditional covering arrays for each value of t , 2 through 4. Since our algorithms make some random decisions in generation, the sizes of final arrays vary.

Figure 5 compares the number of configurations at each array strength, the number reused in the incremental approach, and the total number of configurations in each specific test schedule. The left part of this figure shows the traditional approach while the right side shows the incremental approach. Each of the incremental parts is labeled with a letter to show how the final arrays are generated. The numbers of “new” test cases that must be run at each strength is given below the arrays. As can be seen if we use the incremental approach, starting at 2-way and successively increasing to 4-way for this model, then we need to run 834 configurations (68 2-way + 180 3-way + 586 4-way). This compares favorably with using a traditional 4-way covering array where we need to run 832 configurations. It is also

Compile Time Options	
Binary Options (Enabled/NULL)	
assembler, local-infile, thread-safe-client, archive-storage-engine, big-tables, blackhole-storage-engine, client-ldflags, csv-storage-engine, example-storage-engine, fast-mutexes, federated-storage-engine, libedit, mysqld-ldflags, ndbcluster, pic, readline, config-ssl, zlib-dir	
Non-binary Options	Values
extra-charsets	–with-extra-charsets=all, –with-extra-charsets=complex, NULL
innodb	–with-innodb,–without-innodb, NULL
Run Time Options	
transaction-isolation	READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, SERIALIZABLE, NULL
innodb_flush_log sql-mode	0,1,2, NULL ANSI,TRADITIONAL, STRICT_ALL_TABLES
Constraints	
innodb requires: transaction-isolation and: innodb_flush_log	–without-innodb NULL NULL
libedit cannot occur with readline	Enabled Enabled

Table 2: MySQL configuration options

considerably more efficient (27%) than using the traditional approach starting at $t = 2$, then running $t = 3$ and then $t = 4$. Under this scenario, we would run 1136 configurations.

Process. For each release we run traditional test schedules in two different ways. First we run a schedule that uses a single strength t . This mimics the traditional approach where we must select a priori which strength t to use. Second we run traditional schedules that begin at a specified t , incrementing to higher strengths as time allows. This mimics a situation in which developers complete a low strength test schedule, but find they still have more time available for testing. For the incremental approach we start at $t = 2$, incrementing as we go until we run out of time.

For both techniques, we run two variants. The first one uses a single array at every given strength. The second uses 3 arrays at every strength. For the incremental approach this corresponds to Figure 3 and Figure 5, respectively. We run the test schedule in the order that completes specific covering arrays the fastest. For instance, if we examine Figure 5 we would first run A, followed by B followed by E followed by C and then F and D.

We ran all of the possible schedules to completion. For our analysis however, we limit the data to the part of the schedule that could have been run during the *actual* lifetime of the release. We compute this by using an average cost of 3 hours to test each configuration, and by assuming a computing grid containing 50 nodes.

Metrics. We use all of the covering arrays from the full traditional approach (3 at each strength of t , from $t = 2$ to $t = 4$) as a baseline for our classification and call this our oracle. It is the best we could have done using the traditional approach. We note that most of our F-measures (the weighted harmonic mean of precision and recall in the classifications) were very close to 1.0 for this data mean-

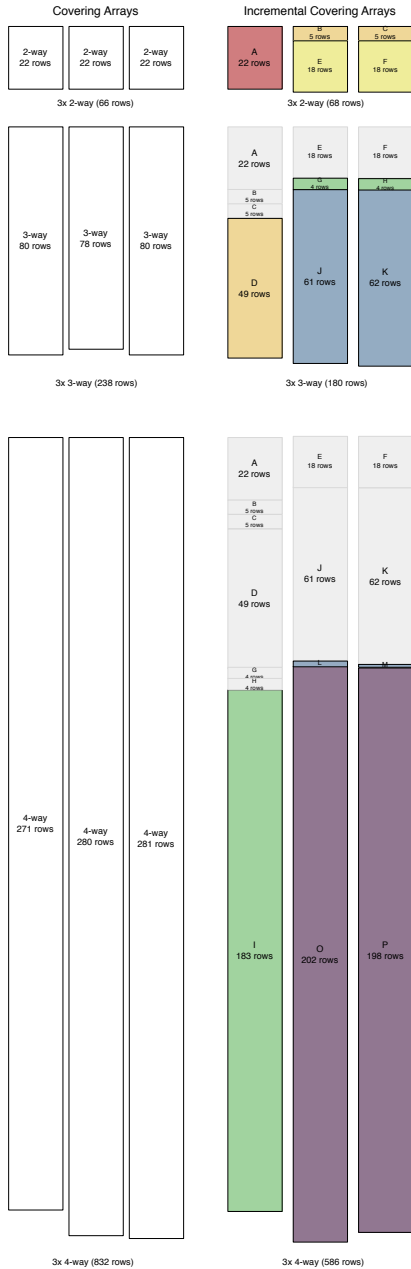


Figure 5: Comparison of covering array schedules

ing that the classifications appear to accurately predict test outcomes.

We took several steps to simplify the data analysis. First, we computed 105 equivalence classes of failure patterns across all of the tests and releases. We formed these equivalence classes by grouping all test cases whose outcomes matched exactly across all configurations in the oracle data set. Next we restricted our analysis to a single representative from each equivalence class. This is appropriate because each of these test cases in a given equivalence class will yield the same classification tree.

For each generated classification tree we computed a *characterization number*. This is the length of the longest path in the classification tree ending at a leaf labeled, “fail”. It reflects that the tree’s characterization that the failure is

caused by a certain number of configuration options.

We note that 36 of 105 test cases could not be classified in terms of MySQL’s configuration options. These test cases are given a characterization number of “0”. For this experiment, this situation occurs in two cases. One is when we don’t observe at least one pass and at least one failure for a given test case. For example, some tests failed every time they ran (because tests can skip certain configurations these don’t all map to a single equivalence class). A classification tree for this data, of course, would not associate the failure with any particular option. The other situation occurs when the test failed in less than 1% of the configurations making up the oracle data. We established this cutoff both to simplify analysis and because such rarely occurring failures are easily confounded with transient failures and are often not related to configuration options. Figure 6 shows the *characterization numbers* for the 105 patterns. The y-axis represents the characterization number.

Next for each proposed covering array regime we executed its test schedule and computed classification trees for each representative test using all configurations in the schedule. For each test case and tree we compared its characterization number to the corresponding one produced using the oracle data. For example, if a test case is characterized by the oracle data as involving 2 configuration options, but is characterized by another approach as involving only 1 option, then the characterization numbers do not match. We used the total number of matching characterization numbers across all test cases as a rough figure of merit for a proposed covering array regime. Intuitively, the more agreement between the more comprehensive oracle data and that generated by the proposed regime, the better the proposed regime is. As discussed later, we will also do detailed manual comparisons of the actual failure characterization where more detail is required.

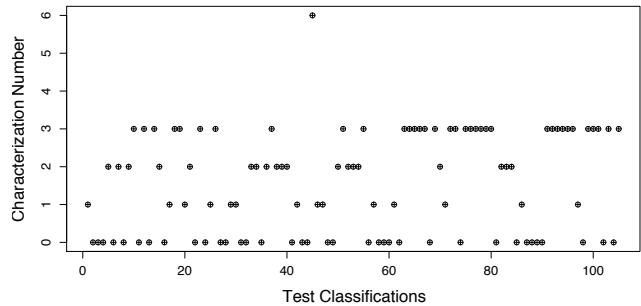


Figure 6: Characterization numbers for rel. 1

Threats to Validity. All experiments suffer from threats to validity. In this study we have used a single software system, with many native failures, and a large configuration space. Although MySQL shares some characteristics with other large configurable systems, the types of failures it experiences may differ. In our experiments we tried to select a broad configuration sample, but due to the sheer size of the entire configuration space for the software subject, we were only able to test a small portion. It is possible that choosing a different configuration space might impact the results and alter them. For instance, the failures that we are currently classifying as 2-way failures may indeed have shown themselves to be higher order failures if a larger configuration space was chosen. We do not believe, however, that

t	Num at t	Match (num)	Match (%)	Configs	Run (%)
Oracle					
		105		1136	100
Traditional Approach					
2	1	51	49	22	100
3	1	61	58	80	63
4	1	66	63	271	18
2	3	66	63	66	76
2,3	1	83	79	100	50
Incremental					
2,3	1	82	78	81	62
2	3	80	76	68	74

Table 3: Characterization of rel. 1: 159 min. budget

this will change the results of cost comparison which are independent of the types of failures seen because we have considered several different scenarios.

5.2 Results

In this section we evaluate our approach on three releases of MySQL, each of which has a different lifetime. We begin with release 1.2510 which was current for only 159 minutes. We then examine release 1.2511 which was current for just under 2 days. Finally we examine 1.2512 which was current for about 2 weeks.

5.2.1 Release 1: Lifetime of 159 Minutes

This release, had a short lifetime during which we could test only 50 configurations. With this we could run 2 complete 2-way arrays, but could not run any complete 3-way arrays. Table 3 shows the overall classification data for this build. The rows represent the various schedules tested, while the columns show the number and percentage of characterization numbers that match at each strength as well as the number of configurations in the schedule and the percentage run. As can be seen, the traditional approach for a single strength array performed worse than the incremental approach for this release. For instance when we run a single array at $t = 2$ or $t = 3$ we see match percentages of 49% and 58% respectively while the incremental arrays for $t = 2, 3$ matched 78%. The traditional approach that executes both a 2- and a 3-way array, performed as well as the incremental approach. Also, using multiple 2-way arrays did as well or better than using single arrays at higher strength.

Table 4 shows detailed data for this release. In each cell we show the number of 2-way failures in which the proposed covering array approach and the oracle data yield identical classification trees. This information, for example, appears in the third column of Table 4, the first number is the number of identical trees and the second – in parentheses – is the number of matching characterization numbers. In many instances where the characterization number matched, the underlying trees were identical, but this was not always the case. In particular, the traditional, single 4-way array generated 8 trees with the correct characterization number of 2, but 7 of those trees had incorrect characterizations. This shows one of the dangers of executing too large a test schedule that can't be completed in the available time. In this case only 18% of the test schedule could be completed, re-

t	Num at t	Num Correctly Classified (Characterized)	% Correctly Classified
Oracle			
		19	
Traditional Approach			
2	1	1 (2)	11
3	1	10 (10)	53
4	1	1 (8)	5
2	3	8 (9)	42
2,3	1	9 (10)	47
Incremental			
2,3	1	14 (14)	74
2	3	13 (13)	68

Table 4: 2-way classification matches for rel. 1

t	Num at t	Match (num)	Match (%)	Configs	Run (%)
Oracle					
		105		1136	100
Traditional Approach					
2	1	51	49	22	100
3	1	101	96	80	100
4	1	104	99	271	100
2	3	96	91	66	100
3	3	104	99	238	100
4	3	105	100	832	96
2,3,4	1	104	99	373	100
3,4	1	105	100	351	100
2,3,4	3	104	99	1136	70
3,4	3	104	99	1070	75
Incremental					
2,3,4	1	104	99	272	100
2,3,4	3	103	98	834	96

Table 5: Characterizing rel. 2: 1.9 day budget

sulting in an unbalanced data set from which generalization was difficult.

5.2.2 Release 2: Lifetime of 1.9 Days

This release lasted about 2 days, therefore we could test about 800 configurations, which provided us with richer information. Some schedules still did not finish, but many fewer than in the previous release. Table 5 presents some of the classification results. Here the poorest performer was the traditional, single 2-way array. Most of its difficulties stem from misclassifying 3-way failures as 2-way. This problem is greatly curtailed by using multiple traditional 2-way arrays because the extra data points limit incorrect generalizations. Alternatively, running one or more 3-way arrays would have also improved performance. Most of the other schedules performed well as all but one test case has a characterization number of 3 or less and because there was enough time to complete at least one full 4-way array. The incremental approaches were essentially identical to the best performing traditional approaches. We also note that while running multiple covering arrays at a given strength does not greatly improve overall classification accuracy in this re-

t	Num at t	Match (num)	Match (%)	Confgs	Run (%)
Oracle					
		105		1136	100
Traditional Approach					
2	1	51	49	22	100
3	1	75	71	80	100
4	1	101	96	271	100
2	3	96	91	66	100
3	3	95	90	238	100
4	3	104	99	832	100
2,3,4	1	101	96	373	100
3,4	1	101	96	351	100
2,3,4	3	105	100	1136	100
3,4	3	104	99	1070	100
Incremental					
2,3,4	1	104	99	272	100
2,3,4	3	102	97	834	100

Table 6: Characterizing rel. 3: 2 week budget

t	# at t	2-way		3-way	
		Class. (Char.) Num	Class. %	Class. (Char.) Num	Class. %
Oracle					
		19		35	
Traditional Approach					
2	1	1 (2)	5	0 (1)	0
3	1	15 (16)	79	9 (11)	82
4	1	19 (19)	100	34 (34)	97
2	3	16 (17)	84	25 (29)	71
3	3	19 (19)	100	28 (28)	80
4	3	19 (19)	100	35 (35)	100
2,3,4	1	19 (19)	100	34 (34)	97
3,4	1	19 (19)	100	34 (34)	97
2,3,4	3	19 (19)	100	35 (35)	100
3,4	3	19 (19)	100	35 (35)	100
Incremental					
2,3,4	1	19 (19)	100	34 (35)	97
2,3,4	3	19 (19)	100	33 (33)	94

Table 7: 2 and 3-way classifications for rel. 3

lease, it did prevent misclassifications of a non-deterministic failure. In contrast, several of the traditional single array methods failed to characterize the failure properly because it didn’t manifest itself enough times in the observed data.

As before, the cost of the incremental approach is equal to running a fixed traditional 4-way array. The cost of running the traditional approach incrementally is considerably higher than the cost of using our incremental approach.

5.2.3 Release 3: Lifetime of 2 weeks

This was our longest running test schedule. Here we were able to complete all schedules up through 4-way using each of the methods. The data for this release can be seen in Table 6. This release provides us with a view of the best possible case for each of the schedules. Once again we see that the system classified poorly when it used lower strength traditional schedules. Using multiple arrays at a fixed strength of 2 or 3 was better than using a single one (20-30% improvement). Still all single strength approaches were inferior to combining multiple strength arrays. Again our incremental approach is comparable to the traditional approach at considerably reduced cost.

When we examine the detailed characterization numbers for all of the 2 and 3 way failures (see Table 7). We see once

again that most of the classifications whose characterization number matches the oracle also shares the same classification tree. In Table 7 we provide the number of matching characterization numbers in parentheses. As expected, the greatest variance appears when using a single low strength array as there is little data from which to generalize failure characterizations.

6. RELATED WORK

Other techniques have been used to isolate failures in code for debugging [11, 17]. The bug isolation project, uses code instrumentation and statistical sampling to achieve failure characterization [11], while the delta debugging project isolates minimal subsets of tests that cause failures through successive elimination of the input space [17]. Neither of these projects address the configuration space directly. Covering arrays were used for failure characterization in [16], however this work assumes a priori knowledge about the types of failures and assumes resources are available to run the selected strength arrays. Advice is given to practitioners for selecting the correct strength array to mitigate the issues we raise. Another type of covering array, called a variable strength covering array was also used for characterization. The test schedules generated from these, contain portions of the configuration space that are tested at a higher t than other parts, however the different strengths are still decided statically at the start of testing and do not change.

Covering arrays have been used frequently to reduce the number of inputs [1, 6, 7] or configurations [10] when testing a program; however, other than in [16] their primary purpose has been failure detection, not characterization. Construction techniques to build covering arrays [1, 2, 3, 14] describe seeding of rows of the covering array, but for a different purpose. Seeding has been used either to allow testers to request a set of default configurations [1] or as the basis for specialized constructions that generate smaller covering arrays [3]. The work of Tai *et al.* [14] uses a construction method that builds covering arrays by expanding the factors (i.e., the columns), but the purpose, is to allow for new factors to be added, not to change the strength.

Our approach is unique in that we use covering arrays for failure characterization, but do not require developer expertise or a priori knowledge in setting covering array strengths. Instead we employ an initial lightweight sample and then incrementally build using seeding as a both a construction technique, and as a mechanism to reuse information from already tested configurations.

7. CONCLUSION AND FUTURE WORK

This paper presents an improved algorithm for generating covering array test schedules that reduces costs and improves flexibility by incrementally constructing and executing covering arrays and by carefully reusing tests from lower strength covering arrays to construct higher strength ones. It also presents a large case study in which we apply this new algorithm to test a non-trivial open source software system. Our algorithm successfully addresses several serious limitations of current techniques. Specifically, developers must currently select a single strength for the covering array. In practice, if developers choose too low an initial strength they will need to start the process from scratch at a higher strength. If they choose too high a strength,

they waste resources and delay the arrival of lower strength results. Also, the typical practice of generating a single covering array at a given strength leads to complications when non-deterministic failures appear.

Our approach, incremental covering arrays, leverages information gained in previous test executions to generate future test schedules. This allows developers to choose the lowest practical value for t , usually 2, because there is minimal penalty for starting too low. This also allows developers to move to higher strength covering arrays only if the results or resource availability warrants them. Finally, running multiple covering arrays at each strength can better support identification of non-deterministic faults, while simultaneously providing data for higher strength arrays.

To evaluate this algorithm, we performed a case study in which we compared our new approach, and traditional covering arrays across a configuration space of >72M configurations of MySQL. For each tested configuration we ran nearly 800 test cases. Despite the limitations of our case study, we tentatively conclude that for this data:

- Classification models based on incremental covering array test schedules were no worse than those based on traditional covering array test schedules.
- In the worst case, the incremental approach starting at strength 2 and working up to strength t , was $\sim 5\%$ more expensive than using fixed strength traditional covering arrays of strength t .
- Compared to using traditional covering arrays incrementally, which typically involves running multiple covering arrays from scratch, our approach required up to 27% fewer configurations. It did this with no loss of quality.
- Our incremental approach was able to detect and classify lower strength faults earlier than traditional fixed strength arrays because they execute complete lower strength covering arrays before moving to higher strengths.

Our future work concentrates on validating, as well as refining and generalizing the incremental covering array approach. First, we plan to expand our study to monitor the *live* MySQL continuous build, integration and test process. We will examine a broader configuration than we are currently using and we will replicate this work on additional large configurable software systems as well. Second we plan to examine alternative models for distributing seeds which will allow us to use differing numbers of covering arrays at each level of t . Finally, we are working on a method for automatically determining “when” the algorithm should adapt by moving to higher strength (versus generating more arrays at the current strength).

8. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through award CCF-0747009, by the Air Force Office of Scientific Research through award FA9550-09-1-0129, the Army Research Office through DURIP award W91NF-04-1-0104, and the Office of Naval Research under contract N000140710329.

9. REFERENCES

- [1] 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. on SW Eng.*, 23(7):437–44, 1997.
- [2] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proc. of the Int’l Conf. on SW Eng., (ICSE)*, pages 38–44, 2003.
- [3] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proc. of Int’l Symp. on SW Reliability Eng. (ISSRE)*, pages 394–405, November 2003.
- [4] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Int’l Symp. on SW Testing and Analysis, (ISSTA)*, pages 129–139, July 2007.
- [5] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: Implications for combinatorial testing. In *Workshop on Advances in Model-based SW Testing*, pages 1–9, November 2006.
- [6] 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 Int’l Conf. on SW Eng., (ICSE)*, pages 285–294, 1999.
- [7] 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 Int’l Conf. on SW Eng., (ICSE)*, pages 205–215, 1997.
- [8] S. Fouche, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *Proc. of the 14th ACM SIGSOFT Symp. on Foundations of SW Eng.*, pages 557–560, 2007.
- [9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. of the 24th Int’l Conf. on SW Eng., (ICSE)*, pages 119–129, New York, NY, USA, 2002. ACM Press.
- [10] D. Kuhn, D. R. Wallace, and A. M. Gallo. Sw fault interactions and implications for software testing. *IEEE Trans. on SW Eng.*, 30(6):418–421, 2004.
- [11] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conf. on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.
- [12] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. N. rajan. Skoll: Distributed continuous quality assurance. In *Proc. of the Int’l Conf. on SW Eng., (ICSE)*, pages 459–468, 2004.
- [13] MySQL, 2006. <http://www.mysql.com>.
- [14] K. C. Tai and L. Yu. A test generation strategy for pairwise testing. *IEEE Trans. on SW Eng.*, 28(1):109–111, 2002.
- [15] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.
- [16] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. on SW Eng.*, 31(1):20–34, Jan 2006.
- [17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. on SW Eng.*, 28(2):183–200, 2002.