Towards Incremental Adaptive Covering Arrays

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 creates a need for more intelligent sampling mechanisms to detect and locate failure-inducing dependencies between configurations. Prior work shows that test schedules based on a mathematical object, called a covering array, can be used to detect and locate failures in combination with a classification tree analysis. This paper addresses limitations of the earlier approach. First, the previous work requires developers to choose the covering array's strength, even though there is no scientific or historical basis for doing so. Second, if a single covering array is insufficient to classify specific failures, the entire process must be rerun from scratch. To address these issues, our new approach incrementally and adaptively builds covering array schedules. It begins with a low strength, and continually increases this as resources allow, or poor classification results require. At each stage, previous tests are reused. This allows failures due to only one or two configurations settings to be found and classified as early as possible, and also limits duplication of work when multiple covering arrays must be used.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Fault Localization, Covering Arrays

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 interdependent 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 interdependencies relies, in part, on effective software testing since these dependencies are often uncovered through the discovery of subtle *interaction* faults—faults triggered by specific combinations of configuration parameters.

One proposed approach for interaction testing involves using a sampling strategy derived from computing mathematical objects called covering arrays [1, 4, 6]. 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. [11] integrated this approach into the Skoll system [8], which is a distributed continuous quality assurance (DCQA) environment that allows for highly parallel execution of QA processes. Covering arrays were first used to generate test schedules. Those schedules were then executed in parallel across a grid of computers, the results were then returned to central servers where uncovered faults were automatically classified to help developers find their underlying causes. The results suggested that the covering array test schedules produced better classification models than equivalently-sized random samples and that the process scaled well to large configuration spaces.

The approach, however, has several limitations. First, it depends on developer insight to select the key sampling parameters. In order to reliably classify faults that are caused by t configuration options, samples must be built that test all t-way combinations of these options. This means the tester must know *a priori* what *strength*—value of t—to use. If they set t too large, resources will be wasted, while selecting t too small may result in poor classification. Since systems often have multiple failures with different causes, either (or even both) of these situations is virtually guaranteed. Second, because it is not generally possible to use a portion of a t-way covering array to reliably classify faults caused by fewer than t options, developers must 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 faults are found and classified as early as possible. Third, as testing continues, each covering array schedule is generated independently from all others. There is no mechanism to exploit the fact that some configurations have already been tested. Basically, this approach often runs more tests than necessary, incorrectly correlates failures with configuration parameters, duplicates work, and suffers delays in reporting classification information.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

To addresses the limitations we propose and evaluate an extension to the underlying covering array technique. The proposed approach is both *incremental* and *adaptive*. It begins with light sampling (i.e. sets t to be small) and classifies faults incrementally—providing early results for developers. It then adapts and uses heavier sampling as results and resources indicate. A central theme of this approach is that it lowers the cost of adapting by carefully reusing results from earlier test runs. It thereby efficiently selects and reuses test workloads to isolate and disambiguate failures by increasing the strength of testing when classifications at lower strengths fail. We call the new approach: *incremental adaptive covering array fault characterization*.

In the next sections we first provide a motivating example as context for our new process and then describe the revised covering array incremental adaptation strategy; Next we analyze the approach; After that we compare our approach to to other scheduling policies; and, finally, we present concluding remarks and possible directions for future work.

2. MOTIVATING EXAMPLE

This work is motivated by our efforts to create an automated continuous build, integration and test (CBIT) process for the MySQL database server project [9]. We intended to test a (partial) configuration space with 110k configurations. In addition, since the update frequency for specific MySQL versions can vary (sometimes multiple times a day, sometimes every few days), testing results have short, but varying, useful lifetimes. Finally, our discussions with MySQL developers suggested that fault patterns (i.e., number of interacting options causing a failure) are poorly understood, but clearly vary over time. Given these constraints we decided to explore using incremental adaptive covering arrays to sample this large configuration space. We note that the challenges we faced on this project are similar to those found in a wide variety of production development environments.

3. APPROACH

Two major limitations of traditional covering arrays are (1) the lack of guidance for selecting an initial interaction strength and (2) the need to redo work if the wrong interaction strength is chosen. Schedules using our incremental adaptive approach addresses both of these issues. We begin by testing at the lowest strength (i.e. t=2), and then successively move to higher strengths. At each stage, we reuse already tested configurations, only testing configurations needed to complete coverage at the current strength. This allows classification as early as possible, improving overall testing efficiency. This also allows us to generate test schedules in environments that have unknown time constraints and hardware resources.

Our technique iteratively *seeds* lower strength arrays to create higher strength arrays. Seeding has previously been used in construction of covering arrays to provide a method for testers to include a *default* set of configurations [1]. It has also been used as a construction technique, for the purpose, of generating minimally sized covering arrays [3]. 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. Suppose we can build a 3-way covering array of normal size using an already tested 2-way covering array. This would mean that we can classify all 2-way faults after the configurations required for 2-way results are run. Furthermore, we only have to run approximately the same number of configurations as before to get all 3-way interactions; i.e. we lose none of the benefits of the 3-way array, but gain the advantage of early classification of the 2-way faults. From a practical point of view, this would mean that we only need to run a portion of the higher strength covering array at any point to get complete t-way coverage.

If successful, this approach would provide two big advantages: (1) building higher strength covering arrays from lower strength ones means that we can reuse already tested configurations as we move up in strength, thus it is less critical to choose the right strength a priori; and (2) this approach allows us to classify lower order faults with as few configurations as possible, allowing developers to start debugging and repair earlier in the test process.

Incremental Adaptive Strategy. The incremental adaptive strategy has several steps. First we select a desired degree of replication. This is the minimum number of covering arrays to run at a given strength. Running more than one covering array helps disambiguate non-deterministic or higher order faults. Next, we build a base t-way covering array. We then use this as a seed for building a new $t+1$-way covering array. We then reuse some configurations in this $t+1$-way array as seeds for additional t-way arrays. The idea here is that, by construction, the union of the configurations in the t-way arrays comprises a large portion of a $t+1$-way array. That is, should we decide to run a $t+1$-way array after running some or all of the t-ways, then much of that work is already done. Unlike when just starting with a $t+1$-way array, t-way faults can be classified and developers can begin to fix them as soon as the much smaller t-way arrays have been run. If developers do wish to continue, 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.

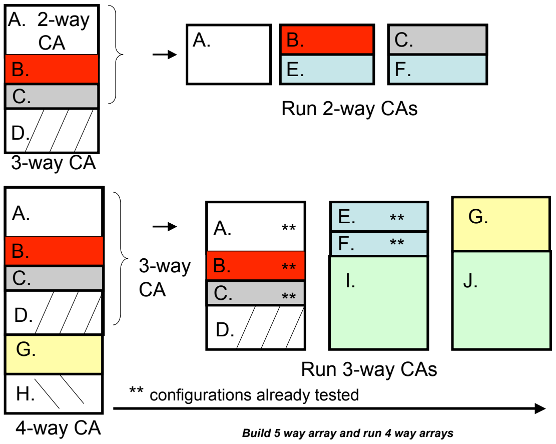

Figure 1: Constructing Iterative CAs

Figure 1 shows this scenario. Here we are running three covering arrays at each strength. An initial 2-way (t) covering array is used as a seed for a 3-way ($t+1$) array. The unseeded part of the 3-way array is split in two ($B\&C$) and (D). The first half is distributed as seeds for the remain-

config. option	values
default character set	binary, ascii, cp1250 cp1257, cp866,gbk greek,hebrew, *latin1* latin7, ujis, utf8
max. indexes	16, *64*, 128
debug level	none, *debug*, full
extra character sets	*none*, all
pthreads	true, *false*
big tables	true, *false*
innodb	*true*, false
archive storage engine	true, *false*
csv storage engine	true, *false*
blackhole storage engine	true, *false*
NDB cluster	true, *false*
federated storage engine	true, *false*
yassl	true, false

Table 1: MySQL configuration parameters

ing 2-way arrays. The remainder (D) will be run when the developer moves to testing at $t = 3$. For each consecutive strength, we use a portion of the seeded t+1 (G) array as well as all of the configurations from the t-1 arrays (E&F) that have not been run. At each step, we always reuse all of the already tested configurations, and run portions of the next strength array. Figure 2 shows the outcome of this method on our study program, MySQL. At each stage, the highlighted portions of the figure show the already run configurations from the previous stage.

4. AN INITIAL ANALYSIS

This section presents some initial analysis of our approach. Our goal is to analyze some of the costs and benefits of the modified approach compared to using the traditional one—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.

Setup. Our subject program for these studies is the MySQL database system [9]. For this study we limited the size of the test configuration space by considering just 13 features (10 binary, 2 with 3 levels, 1 with 12 levels). Table 1 gives the details of our configuration model, with default values in italics. The resulting configuration space contains 110,592 possible configurations. Using the configuration model previously described, we computed both the traditional and incremental covering array algorithms. Figure 2 shows the number of configurations at each array strength, the number actually executed, and the total number of configurations in each specific test schedule.

Results. We now examine how the cost of the traditional approach and the incremental adaptive approach might differ. To do this, we (arbitrarily) examine the process of detecting one failure that happens to be caused by three interacting options. In the first use case, developers will use three traditional covering arrays of strength $t = 4$. In the second use case, developers will, instead, use the incremental approach, starting with an initial strength of $t = 2$ and work forward until they arrive at a final strength of $t = 4$. Using the data generated for the MySQL system, we will compare the work done and time spent to detect this failure under each use case.

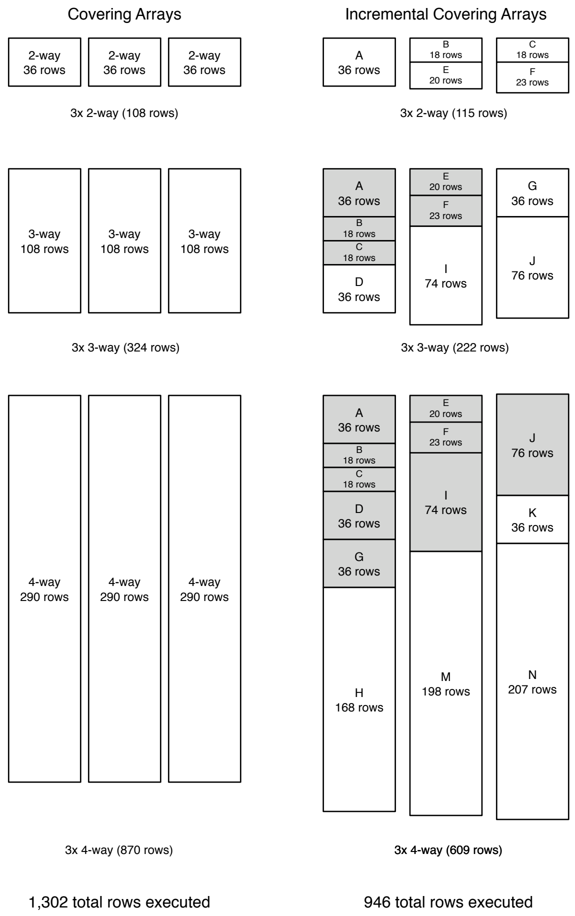

Figure 2: Comparison of Covering Array schedules

Under these assumptions, the traditional approach would have to test 870 configurations. Since each test takes upwards of 2 hours, this requires around 1740 CPU hours in the worst case. The incremental approach would have first run the 115 configurations found in the 3, 2-way covering array schedules. Then it would run another 222 new configurations for the 3, 3-way covering array schedules; and finally another 609 new configurations for the 3 and 4-way schedules for a total of 946 configurations. This requires 1892 CPU hours in the worst case. Here we see that the incremental approach cost about 8.7% more than the traditional approach.

Now we consider the time needed to detect and classify this failure. With the traditional approach all 3-way option combinations must be tested before we can reliably classify the failure. In the worst case, this requires running all 870 configurations in the 4-way schedule. This involves 1740 CPU hours. In contrast, with the incremental approach all 3-way combinations are tested after only 115 + 222 = 337 configurations. Thus, this failure can be reliably classified in about 38.8% of the time required by the traditional approach. Even though our overall cost to run the incremental covering arrays is about 9% higher, we can reliably locate the 3-way fault much sooner.

If, instead, the traditional approach starts at $t = 2$, followed by $t = 3$, we must run 432 (or 28 % more) tests than the incremental approach for reliable classification. In fact, the only scenario in which a 3-way fault is classified earlier by the traditional approach than by the incremental approach, is the one that initially selects the best size for t.

5. RELATED WORK

Other techniques have been used to isolate faults in code for debugging [7, 12]. The bug isolation project, uses code instrumentation and statistical sampling [7], while Zeller and Hildebrandt isolate minimal subsets of fault causing tests through successive input space elimination [12]. Neither of these address the configuration space. Covering arrays are used for fault characterization in [11], but the work assumes a priori knowledge about the types of faults and assumes resources are available to run the selected strength arrays.

Covering arrays have been used frequently to reduce the number of inputs [1, 4, 5] or configurations [6] when testing a program, however, other than in [11] their primary purpose has been fault detection, not localization. Construction techniques to build covering arrays [1, 2, 3, 10] 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.* [10] 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 strength.

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

6. CONCLUSION AND FUTURE WORK

This paper presents an improved approach for generating covering array test schedules that reduces costs by carefully reusing tests from lower strength covering arrays to construct higher strength ones. It also presents a small case study in which we examine this approach on part of a large, open source software system. Our approach successfully addresses several 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 of 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 or higher order failures appear.

Our approach, incremental adaptive covering arrays, addresses these limitations by leveraging information gained in previous test executions to generate future test schedules. This allows developers to choose the lowest practical value for t, usually 2 and to move to higher strength covering arrays only if the results warrant. Finally, running multiple covering arrays at each strength better supports identification of non-deterministic faults, while simultaneously providing data for higher strength arrays.

To evaluate this approach, we compared it to traditional covering arrays on a configuration space of 110k configurations of MySQL. Based on these results, we tentatively conclude that for this data, the approach incurred little extra cost, while allowing faults to be detected much earlier than with traditional approaches.

Our future work concentrates on validating, as well as refining and generalizing the incremental adaptive covering array approach. First, we plan to expand our case study to a set of controlled experiments that will examine a broader configuration space of the current subject, as well as to replicate it on additional large configurable software systems. Second we are working on a method for automatically determining "when" the algorithm should adapt and move to higher strength. Finally, we are working on an algorithm for the seed distribution method so that it can be automated to work across a broad range of parameter values.

7. 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 Transactions on Software Engineering*, 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 Conference on Software Engineering*, 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 *14th IEEE Int'l Symposium on Software Reliability Engineering*, pages 394–405, November 2003.

[4] 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 Conference on Software Engineering*, pages 285–294, 1999.

[5] 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 Conference on Software Engineering*, pages 205–215, 1997.

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

[7] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.

[8] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proc. of the Int'l Conference on Software Engineering*, pages 459–468, 2004.

[9] MySQL, 2006. `http://www.mysql.com`.

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

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

[12] A. Zeller and R.Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.