Feedback Driven Adaptive Combinatorial Testing

Emine Dumlu and Cemal Yilmaz
Faculty of Eng. and Nat. Sci.
Sabanci University
Istanbul, Turkey
{edumlu,cyilmaz}@sabanciuniv.edu

Myra B. Cohen
Dept. of Comp. Sci. & Eng.
University of Nebraska-Lincoln
Lincoln, NE 68558
myra@cse.unl.edu

Adam Porter
Dept. of Comp. Sci.
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

ABSTRACT

The configuration spaces of modern software systems are too large to test exhaustively. Combinatorial interaction testing (CIT) approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations. The basic justification for CIT approaches is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. We conjecture, however, that in practice many such behaviors are not actually tested because of *masking effects* – failures that perturb execution so as to prevent some behaviors from being exercised. In this work we present a feedback-driven, adaptive, combinatorial testing approach aimed at detecting and working around masking effects. At each iteration we detect potential masking effects, heuristically isolate their likely causes, and then generate new covering arrays that allow previously masked combinations to be tested in the subsequent iteration. We empirically assess the effectiveness of the proposed approach on two large widely used open source software systems. Our results suggest that masking effects do exist and that our approach provides a promising and efficient way to work around them.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation

Keywords

Combinatorial testing, adaptive testing, covering arrays, software quality assurance

1. INTRODUCTION

Software customization, through the modification of runtime or compile-time preferences, allows users to make controlled variations to how their software behaves. Customizable systems such as web servers (e.g. Apache), databases (e.g. MySQL), application servers (e.g. Tomcat) or office applications (e.g. MS Word) which have dozens or even hundreds of customizable options can have an enormous number of configurations.

While validating the correctness of the system across its entire configuration space is desirable, exhaustive testing of all configurations is generally infeasible. One solution approach, called combinatorial interaction testing (CIT), systematically samples the configuration space and tests only the selected configurations [2, 7, 9, 11, 18].

CIT approaches generally work by first defining a model of the system's configuration space – the set of valid ways it can be configured. Typically, this model includes a set of configuration options, each of which can take on a small number of option settings. Given this model, CIT methods next compute a small set of concrete configurations, a t-way covering array, in which each possible combination of option settings for every combination of t options appears at least once [7]. Finally, the system is tested by running its test suite on each configuration in the covering array.

Covering array approaches generally assume that there are no unknown control dependencies among the configuration options, option setting combinations that effectively cancel other options setting combinations. Known control dependencies are worked around by specifying constraints [7, 8] or by defining a set of default test cases in addition to the covering array [7]. Given these assumptions, and assuming the existence of a well constructed test suite, the basic justification for covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options.

We hypothesize however that in practice many such behaviors are not actually tested due to *masking effects*. That is, we believe that some test failures can perturb program execution in ways that prevent other behaviors from being tested. Moreover, we believe that masking effects are not accounted for with current test processes. As a result, developers may develop a false confidence in their test processes, believing them to have tested certain option setting combinations, when they in fact have not. One simple example of a masking effect would be an error that crashes a program early in the program's execution. The crash then prevents some configuration dependent behaviors, that would normally occur later in the program's execution, from being exercised. Unless the combinations controlling those behaviors are tested in a different configuration, or unless the error

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada

is fixed and the faulty configuration is re-tested, we cannot conclude that those configuration dependent behaviors have been tested.

Figure 1a (we discuss Figure 1b in Section 3) illustrates masking effects in a hypothetical covering array-based testing scenario. In this scenario, we have a 3-way covering array created for a configuration model with 4 configuration options ($o1$, $o2$, $o3$, and $o4$). Each option takes a boolean value (0 or 1) and there are no inter-option constraints. The configurations are tested using three tests ($t1$, $t2$, and $t3$). Literals P and F indicate a test success or a test failure, respectively. Consider test case $t1$. This test case failed whenever $o1 = 1$. As a result, it is possible that the 3-way option setting combinations for options $o2$, $o3$, and $o4$ that appear with $o1 = 1$ in the 4 failing runs were not actually tested. In fact, as these 4 combinations appear nowhere else in the covering array, it's possible that they were never tested at all. In this case, a solution could be to set $o1 = 0$ in each of the failing configurations and to rerun the test case. For more complex examples, where the failure is caused by more than one option setting combination and where there are multiple failures, a more complicated response may be necessary.

In this work we present a feedback driven adaptive combinatorial testing approach to prevent the harmful consequences of masking effects. At each iteration, we detect potential masking effects, isolate their likely causes, and then schedule the set of t-way option combinations that are being masked for testing in the subsequent iteration. The process iterates until for all tests each and every t-way option setting combination is present in at least one configuration in which the test passed or failed with a non-option-related cause, or the combination is marked as failure inducing. Our empirical evaluation, conducted on two large and popular open source software systems, suggests that the proposed approach is better than other approaches in preventing masking effects.

The remainder of this paper is organized as follows: Section 2 briefly presents some background information and discusses the related work; Section 3 describes the proposed approach; Section 4 describes the empirical studies; and Section 5 presents concluding remarks.

2. BACKGROUND AND RELATED WORK

Covering array based sampling for software testing, is a specification-based technique that was originally proposed as a way to ensure even coverage of combinations of input parameters to programs [2, 5, 7, 9]. In more recent work covering arrays have been used to model configurations that should be selected for testing [10, 15, 18], where the covering array defines a *test schedule* and each configuration is tested with an entire suite of test cases. Some other domains for which covering arrays have been used in testing is to test graphical user interfaces [19] and in model based testing [4].

A covering array [7] is an array of size $N \times k$ where N defines the number of configurations to be tested and k is the number of configuration options that can be manipulated. Each of the configuration options will have some number of settings (often denoted as v or v_1 when different configuration options have differing numbers of settings). Each configuration (or row of the array) is a set of valid settings, one for each configuration option, of the software under test. Within the set of N configurations, each t-way combination of option settings will be found *at least* once, where t , the test strength, is usually much smaller than k, with t=2 as the most common strength. Empirical research has shown that $t < 6$ can potentially find a large proportion of interaction faults [11]. An interaction fault is one that can only be manifested when a specific set of t settings is used in the same configuration. Further empirical results show that covering arrays are effective in practice [10, 15, 16, 18].

The core of a covering array is the model which defines it, that includes k, each of the v_i's, and t as well as any dependencies between option settings. Configuration options that are modeled may be controlled at both run-time or compile-time [18]. While both important in the model, failures may impact the results of testing in different ways. For instance if a compile-time option fails to build, we cannot run even a single test, whereas if a run-time option fails on one test, it may succeed on another.

A recent focus for research on covering arrays has been the study of the impact of *inter-option constraints*, dependencies between specific settings [3, 6, 8]. For instance, the presence of setting *on* for one configuration option, may force another configuration option to take a setting of *off*. This changes how many valid combinations of t-way settings there are, and may impact the number of configurations needed for testing. Constraints may also be responsible for masking effects as we see in our experiments. We use the notion of constraints in this work to encode masking-combinations as to be avoided.

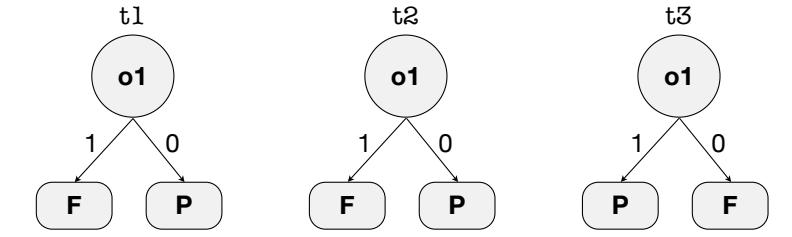
The work in this paper uses classification trees to characterize faults (see the next section) based on the pass/fail results of the covering array test schedules. This follows along the lines of a previous work of us in [18]. In addition, in [10] we developed an incremental method for building covering arrays which provides a way to reuse results between test schedules, building higher strength covering arrays from lower strength ones that can be used when there is little a priori knowledge of testing time and resources. In both of these papers classification is important for fault characterization. Our use of classification in this paper is different. We use it to identify the masking effects so that they can be prevented; e.g. it is not the end result but part of the process.

In specification based testing, there has been considerable work on modeling partitions of a system, through techniques such as the category partition method or the Test Specification Language, TSL [14]. In category partition, one first identifies the parameters and their choices. Once this has been achieved then several types of constraints are added. Some choices may be dependent on others which become dependency constraints, while others choices must be tested alone and are flagged as `error` or `single` and are never combined with other option settings. This is similar to one part of our masking problem, yet the constraints in category partition are manually discovered.

In prior work on using covering arrays for testing configurations [15, 18], the system under test has one configuration model and each configuration runs the same set of test cases. The work in this paper differs in that it creates a configuration model for *each test case* and schedules potentially different sets of test cases to be executed in each configuration. We do not know of other work that uses this notion of a per-test configuration model.

o1	o2	o3	o4	$t1$	t2	$t3$
1	1	1	1	F	F	P
1	1	0	0	F	F	P
1	0	1	0	F	F	P
1	0	0	1	F	F	P
0	1	1	0	P	F	F
0	1	0	1	P	P	F
0	0	1	1	P	P	F
0	0	0	0	P	P	F

(a)



(b)

Figure 1: Classification models created for an example scenario.

```
# test-specific constraint
o1!=1

# seed
o1 o2 o3 o4
 0  1  1  0
 0  1  0  1
 0  0  1  1
 0  0  0  0
```

(a)

```
# test-specific constraint
o1!=0

# seed
o1 o2 o3 o4
 1  1  1  1
 1  1  0  0
 1  0  1  0
 1  0  0  1
```

(b)

o1	o2	o3	o4	tests
0	1	1	1	{$t1$, $t2$}
0	1	0	0	{$t1$, $t2$}
0	0	1	0	{$t1$, $t2$}
0	0	0	1	{$t1$, $t2$}
1	1	1	0	{$t3$}
1	1	0	1	{$t3$}
1	0	1	1	{$t3$}
1	0	0	0	{$t3$}

(c)

Figure 2: (a) Configuration model for t1 and t2 (happen to share the same model). (b) Configuration model for t3. (c) Covering array computed.

3. FEEDBACK DRIVEN ADAPTIVE COMBINATORIAL TESTING

In this paper, we create and evaluate tools and techniques that attempt to ensure that each test case has a fair chance to test all required option combinations. In short, we want to prevent masking effects from fooling us into thinking that we have tested option setting combinations when we have in fact not.

To do this we first modify the definition of t-way interaction coverage as follows: For a given value of t, the configuration space is covered, *iff*, for all test cases, and for all valid $s \leq t$-way combinations of option settings, the option setting combination was present in at least one configuration in which 1) the test case passed, 2) the option setting combination is designated as a failure cause, or 3) the option setting combination was present in at least one configuration in which the test case failed with a non-option-related cause.

The rationale for this criterion is as follows; when a test case runs successfully in a given configuration, all $s \leq t$-way option setting combinations present in that configuration have been tested with that test case. When we determine that some specific option setting combination is causing a test case to fail, then we have tested that combination, but may not have tested any other combinations present in the configuration. When a test case fails, but we cannot determine an option-related cause, then we have not detected a masking effect and must assume that all option setting combinations present in the configuration have been tested with that test case.

We refer to the new coverage criterion as *tested* t-way interaction coverage. A *masked* t-way combination-test case pair is then defined as a valid t-way combination-test case pair that is not considered to be covered by the tested t-way coverage criterion.

Using this definition of coverage we then implement an automated process that takes a system, its configuration space model, and a set of test cases as input and iteratively attempts to achieve complete coverage under our new criterion.

3.1 Process Overview

At a high level our Feedback-Driven Adaptive Combinatorial Testing process operates as follows:

1. **Generate** a covering set of configurations meeting the t-way interaction coverage criterion.
2. **Execute** each test case on each configuration in the covering set.
3. **Analyze** the test case results to identify possible masking effects.
4. **Compute Coverage** to determine if the tested t-way coverage has been achieved. If not, continue to the next step. Else, finish the process.
5. **Regenerate** new covering sets to test previously masked t-way combination-test case pairs. Return to step 2.

We now present a general summary and rationale for each step in the process. Following this section we present an equivalent, but more detailed algorithmic view of the process.

Step 1: Generating covering sets. At each iteration of our process, we compute a set of t-way combinations, one for each test, to be covered in the current iteration. To reduce testing costs, we would like to cover these combinations

using as few concrete configurations and test runs as possible. Consequently, we compute a traditional t-way covering array for this purpose. One problem with this approach however is that, in practice, each test case can have its own test-specific constraints. For example, each test case may be designed to run in some configurations, but not in others. Thus, constructing a single covering array across all test cases might lead to larger covering arrays than is strictly necessary for many test cases. For instance, in a previous work with MySQL (a widely-used open source database management system), we observed that roughly 250 of about 1000 test cases can only run in certain configurations. In essence, large portions of the configuration space simply don't exist for these test cases.

We were unable to find any existing research or tools dealing with unified covering array generation for multiple test cases[1]. We, therefore, developed a novel approach to handle test-specific constraints in the computation of covering arrays. Our approach uses covering array construction as a computational primitive. In this case, we use a tool, called ACTS [1], but other tools may work just as well.

ACTS takes as input a configuration model. The model includes configuration options, their settings, globally enforced inter-option constraints, and a seed. The seed is a set of configurations fed to the tool. Given a strength of the array (i.e., t), ACTS generates a traditional t-way covering array around the seed. Conceptually, ACTS treats all the valid t-way combinations included in the seed as already covered and generates new configurations to cover the rest of the combinations.

To this we add a separate configuration submodel for each test. This submodel, in addition to inheriting all inter-option constraints that must be enforced globally, includes the test-specific constraints. Furthermore, all t-way combinations of option settings considered to be already covered by the test are expressed as a seed in the model (i.e., all the currently covered interactions). As an example, Figure 2a-b depict the configuration models created for the tests in our running example. We are only showing the test constraints and seeds (the base model can be extracted from Figure 1).

For each test, we feed its configuration model to ACTS. The output is a covering array containing the yet uncovered combinations. We then merge the covering arrays generated for the tests and apply a greedy reduction algorithm to further reduce the number of new configurations to be tested as well as the number of test runs to be performed in the subsequent iteration (Section 3.2).

The output of this process is then a set of final configurations together with the list of the test cases to be executed in each configuration. Figure 2c illustrates a 3-way test-aware covering array computed for the configuration models given in Figure 2a-b.

Step 2: Execute test cases. At each iteration, we execute the tests in the covering set and record their pass/fail result. The test results are then organized in a data table which is structurally similar to the one given in Figure 1a.

One optional step, that we include in this work, is that we treat the process of building the system as a special test, called a *build test*. This test must, of course, run before other traditional runtime tests unless a properly-configured, compiled system is available. For the build test, a passing result means that the system built without any build errors. Fail means that some build error occurred. As with any other regular test, we seek to achieve a complete coverage for the build test.

Step 3: Analyze test case results. Next we analyze the test results to identify the option setting combinations that are causing failures and thus potentially creating masking effects. Since we cannot do this in a fully automated fashion, we instead use a machine learning approach, called classification trees, to automatically identify likely failure causes.

Classification trees use a recursive partitioning approach to build a model that predicts class membership (i.e., passing or failing a test case) in terms of a set of measurable features (i.e., configuration option settings) [12]. For example, we feed the test result data from Figure 1a to a classification tree algorithm, it might generate three classification models, one for each test case, such as those shown in Figure 1b. Non-leaf nodes represent options, edges represent option settings, and leaf nodes indicate expected test results. The simple classification model obtained for $t1$, for instance, tells us that when test case $t1$ runs on a configuration in which $o1 = 1$, the test case can be expected to fail. Otherwise, the test case can be expected to pass.

These simple classification models have only one single leaf node indicating test case failure, but there could be more than one such leaf node in general. In these cases, we can extract all likely failure inducing interactions, by examining each leaf node that indicates a failure. For each such leaf node, we identify the path from the tree root to the leaf and output a logical rule corresponding to the conjunction of option settings found on the path. This rule indicates a set of option setting combinations that when present resulted in the test case failing. Once we have processed all such paths, the set of likely failure inducing option combinations is simply the disjunction of the path rules.

While producing the classification trees, we take several steps to prevent overfitting the data. That is, we try to make sure our classification models are not treating random errors or noise as failure causes. One standard technique we use is to create the classification models using n-fold stratified cross-validation [12]. This approach essentially builds multiple models from different subsets of the input data, and uses the results to identify candidate models that are not overly influenced by a few individual data points.

Finally, for each likely failure inducing interaction we assign a score, called the F1-measure, indicating the success of the rule in predicting failures in the test data. The F1-measure is a well-known metric, which is computed by combining two standard metrics: precision (P) and recall (R). For a given rule U, F1-measure is defined as follows:

$$recall = \frac{\#\,of\,correctly\,predicted\,failures\,by\,U}{total\,\#\,of\,failures}$$

$$precision = \frac{\#\,of\,correctly\,predicted\,failures\,by\,U}{total\,\#\,of\,predicted\,failures\,by\,U}$$

$$F1 - measure = \frac{2PR}{P+R}$$

[1]Existing work tends to view each configuration as input to a single test case, or uses a single covering array across all test cases.

The F1-measure ranges between 0 and 1, inclusive. Figure 1b shows the F1-measures for the rules obtained in our running example. For example, the recall and precision value of the rule obtained for test case $t2$ is $4/5 = 0.8$ (the prediction for the fifth configuration is a false negative) and $4/4 = 1$, respectively. Thus, the F1-measure of the rule is 0.89. The higher the F1-measure, the better the rule is in predicting failures.

For this paper, an interaction is considered likely to be failure inducing, iff, the corresponding rule score is greater than a predetermined value, called the *cutoff*. Any failures that are not explained by a significant failure cause are considered to be non-option-related.

As an example, consider test case $t2$ in Figure 1. The first four failures for this test case occurred when $o1 = 1$. Assuming that 0.89 is above the F1-measure cutoff, then those failures are considered to be option-related (i.e., $o1 = 1$ is a likely failure inducing cause). The fifth failure, however, cannot be attributed to a significant rule, so that failure is considered to be non-option-related.

Step 4: Compute coverage. In this step we compute the coverage to determine if we should invoke another iteration of our process. We remove from further consideration all test cases that have covered all their interactions. If complete coverage has been achieved for each and every test under our coverage criterion, the process exits. Otherwise, we now generate a new covering set and return to step 2.

Step 5: Regenerating covering sets. The output from the previous step gives us several pieces of information for each test case. First, we know each interaction that was covered because the test case passed. These are the interactions present in at least one configuration on which the test case passed. Second, we know each interaction that was covered even though the test case failed. These comprise the interactions present in at least one configuration on which the test case failed and the failure was non-option-related, and those interactions that are likely to be failure inducing. Third, we know each interaction that was potentially masked and therefore remains uncovered.

Using this information, our goal is to generate an updated covering set containing new configurations that cover any currently uncovered interactions. Before doing this we take several preparatory steps. First we identify all newly covered interactions for each test case and add them as seeds to the test-specific configuration models. Next, we take the complement of each newly identified failure-inducing interaction and add them as constraints to each test-specific configuration model. In addition, the test-specific models for runtime tests incorporate any failure inducing interactions stemming from build test failures. Both of these last two steps help the process avoid known failure causes in subsequent test iterations.

Next we examine each test-specific configuration model to determine whether they contain unsatisfiable constraints. Such situations can arise when a test case fails in multiple ways. For example, suppose a given test case fails one way when a particular binary option is true, and fails differently when the same option is false. In this case, our process generates contradictory constraints that no configuration can satisfy. We attempt to accommodate the conflicts in an iterative way by handling only a non-conflicting subset of the constraints at each iteration. We defer the remaining constraints to be handled in subsequent iterations.

Algorithm 1 – adaptiveCA

Input t: Covering array strength
Input *cfgMdl*: Config. model

```
1: fMatrix ← empty
2: knownCauses ← empty
3: currentCA ← computeCA(t, cfgModel)
4: repeat
5:    executeCA(currentCA, fMatrix)
6:    currentCA ← prepareNextRound(t, cfgMdl,
                                   fMatrix, knownCauses)
7: until currentCA is empty
```

Algorithm 2 – prepareNextRound

Input t: Covering array strength
Input *cfgMdl*: Config. model
Input *fMatrix*: Fault matrix
Input *knownCauses*: Likely failure inducing option combinations

```
1: nextCA ← empty
2: for each test τ do
3:    knownCauses_τ ← identifyCauses(fMatrix_τ,
                                     knownCauses_τ)
4:    cfgMdl_τ ← updateCfgMdl(fMatrix_τ, knownCauses_τ)
5:    CA_τ ← computeCA(t, cfgMdl_τ)
6:    CA'_τ ← reduceCA(CA_τ)
7:    nextCA ← nextCA ∪ CA'_τ
8: end for
9: return nextCA
```

As a further heuristic we also developed and experimented with a rule prioritization strategy. In preliminary work we had observed from manual analysis that longer rules were much more likely to give incorrect labelings than shorter rules. Consequently, when using this heuristic we look at all rules produced for a test case in the current iteration and compute the length of the shortest rule. We then proceed with only the rules whose length is the same as the shortest rule. The rest of the rules for the test case are deferred to subsequent iterations.

3.2 Algorithms

Given the previous discussion our basic algorithm should now be easy to follow. Algorithm 1 describes the main *adaptiveCA* routine. *adaptiveCA* loops until no test case is scheduled for testing, making a call to *prepareNextRound* once per iteration.

These routines define and use three main data structures: *fMatrix*, *knownCauses*, and *cfgMdl*. *fMatrix* is a fault matrix that keeps track of all configurations tested, the test cases executed on them, and the test results obtained. *knownCauses* keeps track of all currently known likely failure inducing interactions. *cfgMdl* is a set of configuration models, storing options, their settings, constraints among them, and a seed. When these variables are subscripted with a test case τ, they refer to the test-specific information in the respective data structures. All changes made on the subscripted variables are assumed to be reflected in the original variables.

Looking now at the main *adaptiveCA* routine, we see that it takes an initial configuration model *cfgMdl*, and a strength t, as input. It first initializes the *fMatrix* and *knownCauses*

data structures. At line 3, it creates an initial t-way covering array and enters the testing loop. At line 5, all selected configuration-test case pairs are tested and the results are returned. These test results are then passed to *prepareNextRound*, where they are analyzed and the covering array to be tested in the next iteration is computed (line 6). The loop terminates when the newly computed covering array is empty (line 7), indicating that full coverage under our coverage criterion has been obtained.

Algorithm 2 depicts *prepareNextRound*, which works as follows. For each test case, we first identify the likely failure inducing interactions by using a classification tree algorithm (line 3). To compute the classification model, we create appropriate data files containing all the configurations tested so far, in which the test case either passed or failed with non-option-related cause. The training data is fed to the classification algorithm. Likely failure inducing options are extracted from the resulting classification model and then scored (Section 3.1). Among the likely failure inducing combinations, only the ones that have a score greater than a given cutoff value are selected. The rest are ignored. As mentioned previously, strategies to reduce overfitting and to handle conflicting constraints (Section 3.1) are also implemented at this step.

Next, we update the test-specific configuration model (line 4). To do this we first populate the list of the constraints for the test case with the newly identified test-specific constraints. Note that the list of constraints included in the configuration model of a test case grows monotonically. That is, once a constraint is included in a configuration model, it stays there during the life time of the process. This prevents the process from examining previously identified failing subspaces. We then compute a seed for the configuration model of the test case, indicating the t-way combinations that are considered to be already covered. The seed contains all the configurations tested so far, in which the test case passed or failed with non-option-related cause.

The configuration model of the test case is then fed to the ACTS tool (line 5). The output is a set of configurations in which the test case needs to be executed in the next iteration. Since the seeded configurations may include configurations that are redundant in terms of coverage, we use a greedy algorithm to identify and eliminate redundant configurations at this step (line 6).

Finally, we merge together the covering arrays computed for each test case (line 7) and then return it (line 9). The result is a set of configuration-test case pairs to be tested in the subsequent iteration.

4. EXPERIMENTS

To evaluate our approach we conducted a set of empirical studies. The subject systems for these studies are MySQL (v5.1) and GCC (v4.5.2). MySQL is a database management system; GCC is the GNU's compiler collection. Both systems are publicly-available.

4.1 Experimental Setup

For these experiments, we used the ACTS tool [1] to construct covering arrays. We used Weka's J48 algorithm to build our classification trees, setting the confidence factor to 0.25 and the minimum allowable number of objects in each class to 2 [17]. The classification models were trained and evaluated with 5-fold cross validation and pruning. The cutoff value used for identifying likely failure-inducing interactions was set to 0.8. We used the prioritization heuristic that favors shorter rules over longer rules and we resolved conflicting constraints iteratively (see Section 3). All the experiments were performed on a dual Intel Xeon processor machine with 2GB of RAM, running the CentOS 5.2 operating system. We describe specific metrics and our subjects in their respective study sections.

Table 1: The configuration model of MySQL used in the study. ct and rt stand for compile-time and runtime, respectively.

name	settings	type
asm	{NULL,enable-assembler}	ct
linfile	{NULL,enable-local-infile}	ct
tsc	{NULL,disable-thread-safe-client}	ct
bt	{NULL,with-big-tables}	ct
ec	{NULL,with-extra-charsets=complex, with-extra-charsets=all}	ct
innodb	{with-innodb,without-innodb}	ct
libedit	{with-libedit,without-libedit}	ct
ndbcluster	{NULL,with-ndbcluster}	ct
pic	{NULL, with-pic}	ct
readline	{with-readline,without-readline}	ct
ssl	{NULL,with-yassl}	ct
zdir	{NULL,with-zlib-dir=bundled}	ct
ase	{NULL,with-archive-storage-engine}	ct
bse	{NULL,with-blackhole-storage-engine}	ct
fse	{NULL,with-federated-storage-engine}	ct
trans.-iso.	{NULL,uncommitted, serializable, committed, repeatable}	rt
flush	{NULL, 0, 1, 2}	rt
sql_mode	{strict_all_tables, traditional, ansi}	rt
lp	{NULL,large-pages}	rt
eng-pdown	{on, off}	rt
rbt	{NULL,big-tables}	rt
log-format	{row, statement, mixed}	rt
lb	{skip-log-bin,log-bin}	rt

4.2 Study 1: MySQL Experiments

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). Furthermore, MySQL enjoys a large developer community that actively updates and tests the system.

4.2.1 Study Setup

We created an initial configuration model for MySQL containing 23 options (15 compile-time and 8 runtime). The number of settings for each configuration option varied. We had 18 options with 2 different settings, 3 options with 3, 1 option with 4, and another option with 5 settings (Table 1). The configuration model initially had no constraints.

For our test suite, we used a set of 738 test cases that came with the MySQL source distribution. Each test case has its own oracle which determines whether each test case execution "passed", "failed", or was "skipped". Successful test cases simply emit pass. Failed test cases emit fail and additionally include an error code. A test case returns skipped when it determines that it cannot run on a given configuration. For example, a number of test cases were designed to run only if MySQL is configured with an NDB clus-

Table 2: Proposed approach with t = 2.

iteration	cfgs tested	test runs	lines covered	branches covered	unique errs	unique test-errs	t-masked	testing time	analysis time	total time
1	20	5896	161556	82316	96	1052	525149	266	1	267
2	31	12529	223074	108606	117	1380	278228	818	6	824
3	34	14740	223164	108679	117	1381	242368	1005	10	1015
4	71	17588	223455	108890	118	1388	186919	1604	26	1630
5	121	18599	223621	108967	118	1388	180064	2003	44	2047
6	159	19395	223656	108993	118	1388	179174	2382	74	2456
7	167	19585	223663	108995	118	1388	179166	2455	115	2570
8	178	19753	223669	109004	118	1388	178798	2634	147	2781
9	181	19765	223669	109004	118	1388	178794	2656	247	2903
10	182	19769	223669	109004	118	1388	178794	2665	300	2965

ter support, an in-memory clustered storage engine (i.e., `ndbcluster=with-ndbcluster`). If the current configuration does not support NDB, these test cases will exit immediately returning the skipped result. Classification models built for these test cases, therefore, involve ternary rather than binary classification.

4.2.2 Evaluation Framework

To precisely evaluate the proposed approach, we would need to know the number of valid interactions that never had a chance to get tested with the test cases (i.e., the ones that are masked). We would then need to analyze how much the proposed approach improved on that. However, this would require us to manually identify all the failure inducing interactions and to quantify their masking effects for tens of thousands of failures occurring on hundreds of configurations tested in the experiments. Since this was not feasible, we opted to evaluate the approach only on those failures for which we had a definitive cause and knew exactly what the masking effects were. We, therefore, evaluate the approach on the basis of masking effects caused by build failures and test skips.

We have defined a metric, called *t-masked*. This metric counts the number of unique t-way combination-test pairs that are untested because of build failures or test skips (i.e., the number of t-way combination-test case pairs that are only observed in configurations that failed to build or in configurations that were skipped). If a configuration fails to build, then we know that none of the test cases get to test the valid t-way interactions present in the configuration. Similarly, if a test case skips a configuration, then none of the valid t-way combinations present in the configuration will be tested. The lower the value of t-masked, the better the approach is at removing masking effects. While build failures are real failures, we do not consider test skips to be. To the degree that developers precisely know all the reasons for which each test can skip, they can deal with the issue by introducing test constraints into the covering array construction. Therefore, t-masked will overstate the practical value of our approach. Nevertheless, the effect of not accounting for such test skips is that interactions do not get tested, which is exactly the problem our technique aims to address, and, therefore, the experiment is a useful test of our approach.

In addition to measuring t-masked, we also count the number of source lines and branches covered, as well as the number of unique errors and unique test-error pairs observed. To identify the unique errors and test-error pairs, we analyze the error codes emitted when test cases fail. These metrics give us another way to measure the impact of masking on our testing process.

4.2.3 Data and Analysis

We first ran the process with $t = 2$. Table 2 presents the results. In this table, columns 1-3 indicate the iteration number, number of configurations tested, and number of test runs performed, respectively. The next two columns present the number of lines and branches covered in the application. The next three columns present the number of unique errors, the unique test-error pairs, and the value of t-masked. The last three columns depict the testing time (i.e., the time spent to build the program and run the test cases), analysis time (i.e., the time spent to identify likely failure inducing options and compute test schedules), and the total time, respectively. All the time measurements are given in minutes. Furthermore, all of the numbers reported for an iteration reflect the measurements obtained over all the iterations up to and including the current one.

In this experiment it took 10 iterations to achieve full coverage. The first two iterations addressed two build failures. The test skips were addressed in the remaining iterations. After the first iteration, i.e., after performing traditional 2-way testing, we observed that 55% (525, 149 out of 961, 795) of all valid 2-way combination-test case pairs were actually masked because of build failures or test skips. Among these failures, the process correctly identified a deprecated option setting, `ssl=with-yassl`, that caused 11 out of 20 configurations to fail to build. The logical negation of the combination (i.e., `ssl!=with-yassl`) was then automatically added to the configuration model and a set of 11 new configurations with `ssl=NULL` were computed. Testing these configurations in the second iteration, greatly reduced the number of pairs masked by 47%.

After the second iteration, a failure inducing dependency between **libedit=with-libedit** and **readline=with-readline** was correctly identified. An interesting observation is that although this error was observed in the first iteration, the classification models created then were not able to reveal any statistical pattern. The reason was that all but one of the configurations in the first iteration that had this dependency had already failed because of the `ssl` error. That is, the first failure masked the second failure. However, avoiding the first failure in the current iteration helped us correctly identify the cause of the second failure by making it possible to observe the second failure in more configurations. A total of 3 configurations failed to build because of the error. A set of 3 new configurations with (i.e., `libedit!=with-libedit` $\vee$ `readline!=with-readline`) were scheduled to be tested in the next iteration. Testing them in the third iteration fur-

Table 3: Traditional t-way covering array-based testing.

t	cfgs	test runs	line cov.	branch cov.	errs	test-err pairs	total time
2	20	5896	161556	82316	96	1052	267
3	72	18425	216612	106506	119	1377	1775
4	248	67804	218170	107187	122	1428	4681

ther reduced the number of pairs masked by 13% compared to the previous iteration.

From the third iteration to the last one, since there were no more build failures, the process addressed the test skips. In total, the genuine test constraints for the 225 (76%) of 298 test cases with known constraints were correctly identified. This further reduced the number of pairs masked by an additional 26%.

To understand why we were unable to identify all of the test constraints we conducted a manual analysis. We determined that the reason was that some of the test skips were not deterministic. Some test cases skipped for reasons not related to the configuration – we believe that the real cause was a timeout because the startup took too long.

We observed that the classification models identified the patterns in most of these intermittent skips, but they did so with a lower confidence. One approach to address this issue in the proposed approach is to use a smaller value for the cutoff parameter. For example, had we used 0.6 as the cutoff value in the experiments (instead of 0.8), 94% of the genuine test constraints would have been correctly identified at the end of the fourth iteration.

At the end of the process, the number of 2-way combination-test pairs being masked was reduced by 66% compared to the first iteration. Furthermore, avoiding the masking effects greatly improved the source line coverage by 39%, branch coverage by 32%, the number of unique errors by 23%, and the number of unique test-error pairs by 32%.

These improvements were obtained at the cost of an increased number of configurations and test runs. This is as expected, because, at the end of the first iteration, for example, if one would like to remove the maskings effects caused by the build failures, the only choice he/she has is to select new configurations and run all the test cases in them. An important observation, though, is that much of the improvements were obtained at early stages of the process. For example, at the end of the fourth iteration, the improvements were within less than 4% of those obtained at the end. Had we stopped after this iteration, compared to the last iteration, the number of configurations tested, the number of test runs performed, and the total time required would have been reduced by 60%, 11%, and 45%, respectively. This is important when test resources are scarce and prioritization is needed.

Another observation is that after the fourth iteration, although the process removed new masking effects at each subsequent iteration (except for the last one), the structural code coverage measurements and the number of unique errors and test-error pairs appear to stabilize. We attribute this to the software under test, its test suite, and the configuration model chosen for the study; the tests were given a fair chance to exercise new combinations, but this was not reflected on the metrics observed.

Table 4: Comparing the proposed approach with t = 2 to traditional t-way testing.

approach	2-masked	reduction
proposed approach	178794	n/a
2-way traditional	525149	66%
3-way traditional	240489	26%
4-way traditional	133977	-34%

Comparison with Higher Strength Traditional Arrays. Next, we compared using our approach with $t = 2$ to using traditional 3- and 4-way covering array-based testing. Using a higher strength traditional covering array is the best candidate that we know of to compare our results. For instance, if we use a 3-way covering array to avoid masking effects caused by 1- or 2-way combinations of option settings, we expect that each 1- and 2-way combination will appear multiple times in different 3-way combinations and may avoid the masking. Table 3 summarizes the performance on the traditional testing approaches.

We wanted to know how much higher strength covering arrays help in removing masking effects. Table 4 presents the numbers of 2-way combination-test case pairs masked in the traditional covering array-based testing approaches and in the proposed approach. We observed that, although, using higher strength covering-arrays reduces the unwanted consequences of masking effects, it certainly does not solve the problem entirely.

The last column in Table 4 shows 1 minus the proportion of masked pairs obtained by our approach to masked pairs with the traditional approaches. For instance, our approach had 26% fewer masked pairs than did the traditional 3-way approach. Compared to the traditional 4-way approach, ours had 34% more masked pairs. At the same time, however, our approach covered more total lines and branches and took about 35% less time.

To investigate further we tried running our approach with higher values of $t = 3$, but still monitored the 2-way combination-test pairs. This time we observed that both approaches performed essentially the same ($133,636$ masked pairs for our approach vs. $133,977$ for the 4-way traditional approach).

These results suggest that it will be important to further study the various cost/benefit tradeoffs afforded by our approach. If we allow our approach to run more iterations it may take longer, but reduce masking effects. Instead, using a higher strength covering array to start with might provide smaller reduction in masking, but might run in a shorter amount of time.

Evaluating the approach with t = 3. Finally, to evaluate the approach on a larger value of t, we reran the study with $t = 3$ and monitored the 3-way combination-test case pairs. Table 5 depicts our results. It turned out that 44% of the $15,144,193$ valid 3-way combination-test case pairs were masked in traditional 3-way testing (i.e., the first round of the process).

Our approach took four iterations. This was less than the 10 iterations we had with $t = 2$. The reason for this was that the larger data sets we ran at each iteration, allowed us to discover constraints more quickly. We observed that at each iteration, the process reduced the number of 3-way combination-test case pairs masked. All of the option combinations causing build failures and 61% of the combinations causing test skips were correctly identified. Had we used 0.6

Table 5: Proposed approach with t = 3.

iteration	cfgs tested	test runs	lines covered	branches covered	unique errs	unique test-errs	3-masked	testing time	analysis time	total time
1	72	18425	216612	106506	119	1377	6670639	1774	1	1775
2	108	44957	223500	127900	125	1438	3365540	3909	6	3915
3	230	51626	223719	128001	125	1440	2915475	6183	128	6311
4	315	52171	223725	128010	125	1440	2797589	6869	152	7017

Table 6: Comparing the proposed approach with t = 3 to traditional t-way testing.

approach	3-masked	reduction
proposed approach	2797589	n/a
3-way traditional	6670639	58%
4-way traditional	2970152	6%

as the cutoff value, 96% of the constraints would have been correctly identified.

By the fourth iteration our approach reduced the number of 3-way combination-test case pairs by 58% relative to the first iteration. Source line coverage and branch coverage improved by only 3% and 2%, respectively. We, furthermore, identified 5% more unique errors and 7% more unique test-error pairs. Compared to the 4-way traditional testing, the proposed approach reduced the number of 3-way combination-test case pairs masked by 6% (Table 6).

In summary, for the subject application, its test suite, and the configuration model chosen, we observed that our approach was able to accurately identify masking effects and to generate new configurations that improved overall coverage. In particular, our approach always significantly increased coverage as it progressed. Had we simply stopped after the first iteration as most current approaches do, significant number of interactions would have remained silently untested.

At the same time our approach incurs costs and must be compared against alternative approaches. For instance, using sufficiently higher strength traditional covering arrays reduces masking to some degree, but at greater fixed costs.

4.3 Study 2: GCC Experiments

One important observation we made in the previous study is that much of the observed improvements came from the test cases in which multiple different interactions caused different failures. This occurs when a test case exercises multiple option-related defects present in the program. In this study we explore this scenario with a somewhat artificial test scenario. For this study we use GCC as our subject application.

4.3.1 Study Setup

We first studied the command line options of GCC and identified a small, but quite problematic configuration subspace. Table 7 depicts the 14 binary options used in the study. The `NULL` setting indicates the absence of the respective command line option. The first three options (`help`, `version`, and `targetversion`) are quite interesting, since the presence of any one of these options makes GCC quit the compilation right after printing either a help message or some version information. In effect, no compilation at all is performed. Similarly, the next three options (`syntax`, `preprocess`, and `assemble`) make GCC compile the code

Table 7: Selected command line options of GCC.

option	settings
help	{NULL, –help}
version	{NULL, –version}
targethelp	{NULL, –target-help}
assemble	{NULL, -S}
preprocess	{NULL, -E}
syntax	{NULL, -fsyntax-only}
mips1	{NULL, -mips1}
mips2	{NULL, -mips2}
mips3	{NULL, -mips3}
mips4	{NULL, -mips4}
mips32	{NULL, -mips32}
mips32r	{NULL, -mips32r}
mips64	{NULL, -mips64}
mips64r	{NULL, -mips64r2}

up to a certain stage and then GCC exits. `syntax` checks the code for syntax errors, but does not do anything beyond that. `preprocess` stops the compilation right after the preprocessing step; no compilation proper is performed. `assemble` stops the compilation right after the stage of compilation proper; no assembling is performed. The remaining options (`mips*`), on the other hand, are all platform-dependent options. When they are not supported on a given platform (as is the case in our experiments), GCC quits right away with an error message.

We then created a single test case and its oracle. The test case makes GCC build itself. For the sake of the study, the test oracle regarded a test run in which GCC builds itself in full as a successful run. Any other outcome is considered to be a failure. The oracle was able to categorize the failures into 14 classes (one class for each failure inducing option setting).

Note that the correctness of each option present in the configuration model should be validated via testing and can be handled in some cases by using single configuration test cases [14]. However, the presence of any one of them creates a masking effect, since it prevents the test from exercising the remaining option settings, thus the remainder of the system. Consequently, out of 2^{14} configurations, there is only one configuration that actually builds GCC in full; the one in which all the options take the setting of `NULL`. We refer to this configuration as the *golden configuration*.

4.3.2 Data and Analysis

We ran our process with $t = 2$. Table 8 presents the results we obtained. Our approach achieved complete coverage in 13 iterations. Our analysis revealed that, for each iteration but one, one failure inducing option-setting pair was correctly identified. In the last iteration though, no such pair was revealed because the coverage criterion had been reached.

Note that although one failure inducing option setting was identified at each iteration and it was successfully avoided in the subsequent iteration, the improvement was not reflected

Table 8: Approach with t = 2 on GCC.

iteration	cfgs tested	lines covered	branches covered
1	7	4652	2546
2	11	4693	2567
3	15	4693	2567
4	19	4693	2567
5	22	4693	2567
6	25	4693	2567
7	29	4693	2567
8	32	4693	2567
9	35	4693	2567
10	38	14636	7931
11	41	143939	106554
12	44	143939	106554
13	45	143939	106554

on the structural code coverage measurements up until the tenth iteration. The reason was, since all the configurations except for the golden one were destined to fail, even though a failure inducing setting was avoided by computing and testing a new set of configurations, the newly generated configurations failed. This prevented the test from exercising the code to the fullest extent possible.

The process, however, happened to hit the golden configuration at iteration 11, after correctly identifying and fixing 10 failure inducing option-setting pairs. At this iteration, the source line and branch coverage were improved by 31 and 42 times, respectively. Had we carried out a traditional 2-way covering array-based testing (i.e., had we stopped after the first iteration), any chances for improvement would have been lost. The code coverage measurements stayed the same in the rest of the iterations, since the maximum coverage that the test could have achieved was already obtained at iteration 11.

One further observation is that the process stopped after identifying 12 out of 14 failure inducing option settings. An in-depth analysis revealed that the classification algorithm was not able to expose the remaining faulty option settings due to the limited amount of data present for analysis. After iteration 12, the entire configuration space was reduced to only four configurations, i.e., the exhaustive combinations of the remaining two options. At the end of the last iteration, all of these configurations happened to be tested already. One of them was the golden configuration. The remaining three configurations were marked as failures by using two different labels. However, the classification algorithm was not able to reveal any pattern; a common issue with data mining approaches caused by insufficient amount of data for analysis. In such situations where the configuration space is reduced to the point so that exhaustive testing is feasible, the entire space can be scheduled for testing in the next iteration to exploit the chance of potential improvements.

We also repeated the experiments with $t = 3$ and obtained similar results. The initial 3-way covering array had 18 configurations. The process stopped after 12 iterations, throughout which a total of 101 configurations were tested. The golden configuration happened to be tested at the last iteration, in which similar code coverage improvements were observed.

Finally, we compared our results to those of traditional t-way covering array-based approaches. Table 9 presents the results. Our approach with $t = 2$ and $t = 3$ revealed all 14 types of failures as well as the golden configuration, whereas

Table 9: Comparing the proposed approach to traditional t-way testing in GCC experiments.

approach	cfgs	uniq. errs	1-masked	2-masked	3-masked
2-way	7	3	15	182	n/a
3-way	18	5	6	108	790
4-way	40	6	6	81	611
5-way	133	6	6	72	440
ours t=2	45	14	0	0	n/a
ours t=3	101	14	0	0	0

the traditional testing (including the 5-way covering array-based testing) revealed at most 6 types of failures. None of the traditional arrays created for the study revealed the golden configuration.

We then computed the number of 1-, 2-, and 3-way option setting combinations masked. There were a total of 28 1-way, 238 2-way, and 1232 3-way valid combinations that could be exercised. As the table indicates, the proposed approach (with $t = 2$ and 3) removed all the masking effects and exercised all the t-way combinations that could be exercised. On the other hand, the traditional 5-way covering array-based testing, for example, was not able to exercise even the settings of each and every option. The reason was that all the statically chosen configurations failed due to one option setting or another. On the other hand, the proposed approach, in a feedback-driven manner, identified the cause of masking effects and removed them iteratively.

4.4 Threats to Validity

We have identified several threats to validity for these experiments. First, we have only studied two software systems. This may impact the generality of our results. However, both GCC and MySQL are widely-used non-trivial applications with large configuration spaces and both have been used in other related works in the literature.

For our experiments we selected a cutoff value of 0.8. As discussed, if we chose a lower value (e.g., 0.6), we may have found more constraints, but we did not experiment exhaustively with this parameter tuning and leave this as future work.

Finally, we do not remove constraints during experimentation; as we find new failure inducing option combinations we continue to add them to our configuration model. In a real testing environment it is possible that at some point the defect causing a masking effect is fixed, but in this work we do not examine that scenario. However, we believe that the scenarios studied in this work are realistic, since long times to defect fixes is common.

5. CONCLUDING REMARKS

The basic justification for combinatorial interaction testing approaches, such as covering arrays, is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. We conjecture however that in actuality many such behaviors may not be tested, because of masking effects caused by failures.

To address masking effects, we developed a feedback driven combinatorial testing approach. Instead of statically computing the covering arrays, we compute them in an iterative manner aimed at identifying masking effects and removing them from the covering array construction process in order to better meet our interaction coverage goals. At each it-

eration of the process, we execute test cases, analyze the results, detect potential masking effects by identifying likely option-related causes, and then generate new covering arrays that avoid the likely failure causes while covering previously masked interactions. The process iterates until for all tests each and every t-way option setting combination is present in at least one configuration in which the test passed or failed with a non-option-related cause, or the combination is marked as failure inducing.

We then evaluated this new process by conducting two empirical studies. These studies used two large open source software systems, MySQL and GCC as subject applications. We observed that the proposed approach always significantly reduced the number of t-way combination-test case pairs masked compared to traditional t-way covering arrays. The number of pairs masked was reduced by 66% when $t = 2$, and by 58% when $t = 3$. In both cases, only 19% of all valid intended combination-test case pairs remained masked.

We, furthermore, observed that, for a given t, the proposed approach generally performed better in reducing the number of t-way combination-test case pairs masked compared to higher strength traditional covering arrays. The proposed approach with $t = 2$ reduced the number of 2-way combination-test case pairs masked by 26% compared to using traditional 3-way testing. When $t = 3$ the approach reduced the 3-way combination-test case pairs masked by only 6% compared to using 4-way testing, but had greater line and branch coverage and ran less number of test cases. Overall, we see a variety of cost/benefit tradeoffs that will need further study.

We think that this line of research is promising. One obvious open issue we intend to pursue is to quantify the prevalence of masking effects in more practical settings. We will also examine alternative machine learning approaches and optimizations for identifying likely configuration-related failure causes. Another interesting goal is to work on further improving the approach by automatically identifying control dependencies among configuration options. This would require us to use successful test runs, in addition to failing runs, for inference.

6. ACKNOWLEDGMENTS

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (FP7-PEOPLE-IRG-2008), by the Scientific and Technological Research Council of Turkey (109E182), and by the US NSF awards CCF-0811284 and CCF-0747009 and AFOSR award FA9550-09-1-0129.

7. REFERENCES

[1] Advanced Combinatorial Testing System (ACTS), 2010. `http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html`.

[2] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, 48(10):960–970, 2006.

[4] R. C. Bryce, A. Rajan, and M. P. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. In *Asia Pacific Software Engineering Conference*, pages 259–268, 2006.

[5] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*, 1998.

[6] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and Proofs, Lecture Notes in Computer Science, 4966*, pages 66–83, 2008.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, 1997.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[9] 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 *Proceedings of the International Conference on Software Engineering*, pages 285–294, 1999.

[10] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 177–188, New York, NY, USA, 2009. ACM.

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

[12] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

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

[14] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31:678–686, 1988.

[15] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–85, July 2008.

[16] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *International Conference on Software Maintenance*, pages 255–264, October 2007.

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

[18] 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.

[19] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *International Conference on Automated Software Engineering*, pages 405–408, 2007.