FEEDBACK DRIVEN ADAPTIVE COMBINATORIAL TESTING

by

EMİNE DUMLU

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of

the requirements for the degree of

Master of Science

Sabancı University

July 2011

FEEDBACK DRIVEN ADAPTIVE COMBINATORIAL

TESTING

APPROVED BY

Assist. Prof. Dr.Cemal Yılmaz (Thesis Supervisor)	[illegible]................................
Assoc. Prof. Dr. Erkay Savaş	[illegible]................................
Assoc. Prof. Dr. Albert Levi	[illegible]................................
Assist. Prof. Dr. Kerem Bülbül	[illegible]................................
Assoc. Prof. Dr. Yücel Saygın	[illegible]................................

DATE OF APPROVAL ....

FEEDBACK DRIVEN ADAPTIVE COMBINATORIAL TESTING

Emine Dumlu
Computer Science and Engineering, MS Thesis, 2011
Thesis Supervisor: Assist. Prof. Cemal Yılmaz

Keywords: Combinatorial Testing, Adaptive Testing, Covering Arrays, Software Quality Assurance

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, 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 effcient way to work around them.

GERİBESLEME GÜDÜMLÜ UYARLAMALI BİRLEŞİMSEL TEST ETME

Emine Dumlu
Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2011
Tez Danışmanı: Yrd.Doç. Dr. Cemal Yılmaz

Anahtar Kelimeler: Birleşimsel Test Etme, Uyarlamalı Test Etme, Örtme Dizileri, Yazılım Kalite Güvencesi.

Özet

Modern yazılım sistemlerinin yapılandırma uzayı ayrıntılı bir şekilde test edilemeyecek kadar geniştir. Örtme dizileri gibi birleşimsel etkileşim test etme yaklaşımları, sistematik olarak yapılandırma uzayını örnek olarak dener ve sadece seçilmiş yapılandırmaları test eder. Birleşimsel etkileşim test etme yaklaşımındaki temel gerekçe, t ya da daha az seçenek ayarlarından kaynaklanan tüm sistem davranışını maliyet-etkin bir şekilde uygulayabilmeleridir. Ancak pratikte çoğu böyle davranışların maskeleme etkilerinden dolayı gerçekten test edilmediğini tahmin etmekteyiz. Bu çalışmada maskeleme etkilerini belirlemek ve bu konu etrafında çalışmaya yönelik bir geribesleme-güdümlü, uyarlamalı, birleşimsel test etme yaklaşımı sunmaktayız. Her iterasyonda, olası maskeleme etkilerini belirleyip, sezgisel olarak onların olası sebeplerini izole etmekteyiz ve daha sonra bir sonraki iterasyonda test edilecek olan önceden maskelenen birleşimlere izin veren yeni örtme dizileri üretmekteyiz. Önerilen yaklaşımın etkinliğini ölçmek için, yaygın bir şekilde kullanılan açık kaynaklı iki tane yazılım sistemi üzerinde yaklaşımımızı değerlendirdik. Sonuçlarımız, maskeleme etkilerinin var olduğunu ve yaklaşımımızın maskeleme etkisi üzerinde çalışmak için umut verici ve etkili bir yol sağladığını öne sürmektedir.

Acknowledgements

I would like to thank my thesis supervisor, Asist. Prof. Cemal Yılmaz for his guidance, and all his support throughout my graduate education and for encouragement in completing the thesis.

I would like to express my special thanks to members of Faculty of Engineering and Natural Sciences of Sabanci University who kindly shared the knowledge and experience with me.

I also thank my family who has been there all along to support me in every aspects of my life.

I would like to thank my office group FENS 2001. I would also thank all my friends in or outside of Istanbul, who support me and encourage me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software customization, through the modification of run-time 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, since configuration spaces are combinatorial spaces that grow exponentially in the number of configurtion options, 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] [3][4][18][26].

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 [2]. 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 [2][31] or by defining a set of default test cases in addition to the covering array [2]. 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.

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

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 widely-used open source software systems (namely MySQL and gcc), suggests that the proposed approach is better than other approaches in preventing masking effects.

1.1 Contributions

The contributions of this thesis can be summarized as follows:

1. Defined masking effects
2. Introduced test case-aware covering arrays
3. Defined a novel interaction coverage criterion to reduce harmful consequences of masking effects
4. Developed a feedback driven adaptive combinatorial testing approach to realize the new coverage criterion in practice
5. Empirically evaluate the proposed approach by using two open source widely-used software applications as our subject applications

1.2 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 briefly presents some background information and discusses the related work; Chapter 3 defines masking effects; Chapter 4 introduces test case-aware covering arrays; Chapter 5 describes a feedback driven adaptive combinatorial testing approach; Chapter 6 describes the empirical studies; Chapter 7 presents concluding remarks, and finally Chapter 7 concludes with some future work ideas.

Chapter 2

Related Work and Background Information

In this chapter, we discuss the related work on covering arrays. We furthermore provide some background information about classification trees which we use to automatically identify likely causes of masking effects.

2.1 Covering Arrays

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][11][16][43]. In more recent work covering arrays have been used to model configurations that should be selected for testing [9][10][3], 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 [20] and in model based testing [15].

A t-way covering array [2] is an array of size $\mathrm{N} \times \mathrm{k}$ 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 [28]. Empirical research has shown that t < 6 can potentially find a large proportion of interaction faults [4]. 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 [9][10][14][3][17][38].

Table 1 presents a 3-way covering array created for a hypothetical software system with 10 binary options (i.e., each option has two levels of settings: 0 and 1). Since this is a 3-way covering array, it contains each possible combination of option settings for every combination of 3 options appears at least once. In options A, B, and C for example, one can find all posible combinations of their settings (i.e., 000, 001,010,011, 100, 101, 110, and 111) in the rectangle drawn on the table. This also holds for all the remaining combinations of 3 options (e.g., ABD, ABE, etc.)

Covering arrays aim to cover all t-way option setting combinations by using the minimum number of configurations. The goal here is to reduce the amount of resources required for testing. For instance, although there are $2^{10} = 1024$ configurations in the configuration space of our hypothetical system, our 3-way covering array has only 13 configurations.

We refer to the type of covering arrays discussed in this section as *traditional covering arrays* in the remainder of the document. In this thesis we compute a novel type of a covering array, called *test-case aware covering arrays*. One way test case-aware covering arrays differ from traditional covering arrays is that they are not just a set of configurations as is the case in traditional covering arrays, but a set of configurations each of which is associated with a set of test cases.

Table 1: An example 3-way covering array

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	0	1
0	1	0	0	0	1	1	1	0	1

2.2 Configuration Model

The fundamental construct used to compute covering arrays is a model, called a *configuration model*. The configuration model typically includes a set of configuration options, their discrete settings, and system-wide inter-option constraints (if any). The number of settings for each configuration option can be varied. In a nutshell, the configuration model defines a valid configuration space for testing. A valid configuration is a combination of option settings that does not violate any constraints. The collection of all valid configurations constitutes a valid configuration space for the system under test. To test a configuration, a given test suite is executed in the configuration.

Configuration models are typically extracted by domain experts or test teams. On the other hand, in this thesis we start with an initial configuration model and through experimentation refine the model as we determine new option setting combinations that cause masking effects.

In the previous studies that use covering arrays for testing configurations [3][10], the system under test has one configuration model and each configuration runs the same set of test cases. The work in this thesis differs in that it maintains 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 case configuration model.

2.3 Seeding

A configuration model can contain a seed. The seed is basically a set of configurations. From the perspective of a covering array generator, all the t-way option setting combinations present in the seed are considered to be already covered by the respective configurations. The generator simply computes a covering array around the seed by adding additional configurations to cover the rest of the required combinations.

From the perspective of developers, the seeding mechanism are typically used to guarantee the testing of certain configurations [2][9][24].

In this thesis, on the other hand, we use the seeding mechanism to force test runs to share configurations as much as possible, which potentially reduces the number of configurations required.

The seeding mechanism in one form or another is supported by many covering array generation tools, such as ACTS [6], AETG [2], PICT [24], SST [25]. We in particular use ACTS to generate test configurations in this thesis.

2.4 Constraints

In a highly configurable system, not all system configurations may be valid [29][30][31][32]. To implicitly express invalid configurations, configuration models contain inter-option constraints. Improper handling of constraints can lead to the generation of invalid test configurations, which in turn can lead to wasted testing.

Table 2: Simple example configuration options

Option	Settings
OS	{XP, Linux}
Browser	{IE, Firefox}
DBMS	{MySQL, Oracle}

To illustrate the necessity of constraints, Table 3 depicts a configuration model for a software system with 3 binary options [1]. In this example, we have an inter-option constraint: If the operating system is Linux, then IE (Microsoft Internet Explorer) can not used as a browser. This is because IE does not support any Linux platforms. In other words, any configurations that has Linux and IE at the same time simply do not exist for the system under test. This constraint, for example, can be expressed as an inter-option constraint in a configuration model as: OS != XP $\rightarrow$ Browser = Firefox.

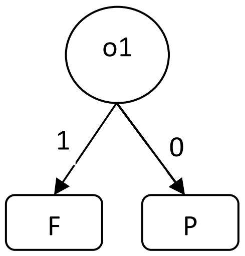
Many approaches have been proposed in the literature to handle inter-option constraints. Cohen et al. study the nature of such constraints in real systems [34] . Mats et al. propose various techniques for efficient handling of constraints [35]. Bryce et al. introduce "soft constraints" to mark option setting combinations that are permitted, but undesirable to be included in a covering array [29].

These approaches are mainly concerned with inter-option constraints, that are enforced globally across the entire configuration space for all the test cases. In the remainder of this document such constraints are referred to as *system-wide inter-option constraints*. In this thesis, we, on the other hand, express options setting combinations that cause masking effects as test case-specific inter-option constraints and enforced them on a per test case basis.

2.5 Methods for Constructing Covering Arrays

The problem of generating covering arrays is NP-hard[44][27]. In the literature, four main types methods have been proposed to generate covering arrays[36][39][40]: greedy methods [2][24], heuristic search-based methods [45][41], mathematical methods [33], and random search-based methods [37][23]. Nie et al. provide a comprehensive survey of these methods [44].

In this work, we use a tool, called ACTS [6][19], that implements a greedy algorithm, called IPOG [42], as a computational primitive to generate test case-aware covering arrays.



$o1 = 1 \rightarrow F$

Figure 1 An example classification tree

2.6 Classification Trees

In this thesis, we use classification tree analysis (CTA) to characterize failing configuration sub-spaces [4][21]. CTA is a recursive partitioning approach to build models that predict a configuration's class (e.g., passing or failing) based on the settings of the options that define the configuration. This model is tree-structured (see Figure 1). Each node denotes an option, each edge represents a possible option setting, and each leaf represents a class or set of classes (if there are more than two classes).

Classification trees are constructed using data called the training sets. A training set consists of configurations, each with the same set of options, but with potentially different option settings together with known class information.

1. For each option, partition the training set based on the settings of that option.
2. Evaluate the option based on how well it partitions configurations of different classes.
3. Select the best option and make it the root of the tree.
4. Add one edge to the root for every option setting.

5. Repeat the process for each new edge. The process stops when no further split is possible (or desirable).

For example, the overly simplified classification tree given in Figure 1 indicates that configurations are likely to fail when the configuration option o1 is 1. Otherwise, configurations are likely to be successful.

To evaluate the classifications, we use it to predict the class of previously unseen configurations. For each configuration, we begin with the option at the root of the tree and follow the edge corresponding to the option setting found in the new configuration. We continue until a leaf is encountered. The leaf's class label is then the predicted class for the new configuration. By comparing the predicted class to the actual class, we estimate the accuracy of the model.

In this research, we analyze the classification trees to extract failure inducing option setting combinations, i.e., the set of options and their settings that are highly corraleted with the manifestation of failures. In particular, we use the Weka implementation of the J48 classification tree algorithm [22] to build classification tree models.

Chapter 3

Masking Effects

Definition. A masking effect is an effect that prevents a test case from testing all t-way option setting combinations present in a configuration, which the test case is normally expected to test.

Table 3: A traditional 3-way covering array vs. a 3-way test-aware covering array

o1	o2	o3	o4	t1
1	1	1	1	F
1	1	0	0	F
1	0	1	0	F
1	0	0	1	F
0	1	1	0	P
0	1	0	1	P
0	0	1	1	P
0	0	0	0	P

Table 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 a test case, t1. 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 (clearly marked in the table) 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.

A harmful consequence of masking effects is that they cause testers to develop false confidence in their testing processes, believing them to have tested certain option setting combinations, when they in fact have not.

Masking effects can be caused by many factors. For example, a software error that crashes a program early in the program's execution could prevent some configuration dependent behaviors that would normally occur later in the execution from being tested. An unaccounted control dependency among configuration options could prevent option setting combinations that are effectively cancelled out due to the control dependency from being exercised. A missing system-wide constraint among configuration options could render a configuration useless; the configuration may not even get compiled.

In this work, we however are primarily concerned with the masking effects caused by option related failures.

Chapter 4

Test Case-Aware Covering Arrays

In this thesis, to reduce the harmful consequences of masking effects we define a novel interaction coverage criterion and then develop a feedback driven adaptive combinatorial testing process to realize the new coverage criterion in practice.

In our automated process we first identify failure inducing option setting combinations that cause masking effects on a per-test-case basis. We then express these combinations as test case-specific constraints in our configuration models so that no configurations with these faulty combinations are tested in the subsequent iterations, effectively removing the masking effects caused by them.

To our suprise, existing covering array-based testing approaches do not provide a systematic way of handling test case-specific constraints. These approaches typically compute a single covering array and then execute all test cases in all of the configurations selected, implicitly assuming that all test cases can run in all configurations. On the other hand, in a feasibility study conducted with MySQL (a widely-used open source database management system) for instance, we observed that roughly 250 of about 1000 test cases do not run when the system is not configured in certain ways. In essence, for these test cases, large portions of the configuration space simply don't exist. Since existing approaches do not take test case-specific constraints into account, they greatly suffer from masking effects.

Table 4: A traditional 3-way covering array-based testing

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

Table 4 depicts a hypothetical 3-way traditional covering array-based testing scenario. Suppose that due to some inter-option dependencies in the code base, t1 and t2 refuse to run in configurations in which o1 = 1 and o1 = 0, respectively. On the other hand, t3 runs successfully in all the configurations. Note that there are 20 valid 3-way option setting combinations to cover for t1 and t2 , and 32 combinations for t3 . Since existing covering array generators do not handle test case-specific constraints, a traditional 3-way covering array was created and all the test cases were executed in all the configurations included in the covering array. As a result, 8 out of 72 (11%) valid 3-way option setting combination-test case pairs were masked due to improper handling of test case-specific inter-option constraints.

Note that expressing test-specific constraints as system-wide constraints in configuration models does not solve the problem. One reason is that constraints for different test cases may conflict with each other. This is indeed the case in our running example; t1 does not run in configurations in which o1 has one setting and t2 does not run in configuration in which the same option has the other setting. Globally enforcing such conflicting constraints would not generate any configurations. Another reason is that even if the test case-specific constraints do not conflict, enforcing them across all the test cases may prevent test cases from exercising some valid combinations, which are invalidated by other test cases. For example, enforcing the test case-specific constraint of t1 on t3 would prevent t3 from testing any combinations in which o1 = 1.

To handle test case-specific constraints in the creation of covering arrays, we define a novel covering array, called *test case-aware covering array.*

In this approach, we take as input a confguration model of the system under test. The configuration model includes 1) a set of configuration options and their discrete settings, 2) a set of system-wide inter-option constraints which are to be enforced globally across the entire configuration space, and 3) a set of test cases together with their test case-specific inter-option constraints which are to be enforced on a per-test-case basis. Given a configuration model and a value of t, a t-way test case-aware covering array is a set of configurations, each of which is associated with a set of test cases such that

1. None of the configurations violate the system-wide constraints.
2. No test case is scheduled to be executed in a configuration that violates the test-specific constraints of the test case.
3. For each test case, each valid combination of option settings for every combination of t options appears at least once in the set of configurations in which the test case is scheduled to be executed.

Table 5: A 3-way test-aware covering array

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

Table 5 presents as an example a 3-way test case-aware covering array created for our running example. None of the test-specific constraints are violated in this test suite. Therefore, no masking effects occur. Furthermore, all valid 3-way option setting combination-test case pairs get to be tested.

As can be seen from Table 5, one way test case-aware covering arrays differ from traditional covering is that they are not just a set of configurations as is the case in traditional covering arrays, but a set of configurations each of which is associated with a set of test cases. Another characteristic of test case-aware covering arrays is that they aim to reduce the cost of testing by running a test case only in those configurations that contribute to the coverage requirements of the test case. For example, in Table 5, each and every test case is scheduled to be executed in half of the configurations included in the covering array; the other half simply does not contribute to the coverage requirement of the test case.

Next, we present an algorithm to compute test case-aware covering arrays.

4.1 Computing Test Case-Aware Covering Arrays

In this approach, we maintain a separate configuration submodel for each test case. The configuration submodel of a test case, in addition to inheriting all system-wide constraints, includes the test-specific constraints.

We first generate a separate covering array for each test case (i.e., for each configuration submodel) in an iterative fashion by using a traditional covering array generator as our computational primitive. We then merge the individual covering arrays created for the test cases to obtain a test case-aware covering array for the entire test suite.

In this work we use a well-known tool, called ACTS [42], to generate traditional covering arrays. However, the proposed approach is readily available to be used with other generators that support seeding.

ACTS takes as input a configuration model. The model includes configuration options, their settings, system-wide 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 t-way covering array around the seed. Conceptually, ACTS treats all the t-way option setting combinations included in the seed as already covered and generates new configurations to cover the rest of the combinations.

Agorithm 1 - ComputeTestAwareCA

Input *t*: Covering array strenth
Input *configModel*: System-wide configuration model

```
1. $ca \leftarrow$ empty
2. for each test $\tau$ do
3.     $seed_\tau \leftarrow$ computeSeed($configModel_\tau$, $ca$)
4.     $ca_\tau \leftarrow$ computeCA($t$, $configModel_\tau$, $seed_\tau$)
5.     $ca_\tau \leftarrow$ reduceCA ($ca_\tau$)
6.     $ca \leftarrow ca \cup ca_\tau$
7. end for
8. return $ca$
```

Algorithm 1 presents the proposed approach. For each test case τ, we first compute a seed (line 3). The seed, out of all the configurations that have been so far included in the covering array ca, contains those configurations that do not violate the constraint of the test case. We then feed ACTS with the seed and the configuration submodel of the test case, $ConfigModel_\tau$ (line 4). The result is a traditional covering array created for the test case at hand. The test case is then scheduled to be executed in all of the configurations selected.

Note that the seed is created to reduce the total number of configurations needed. Since ACTS adds new configurations only to cover t-way combinations that are not already covered by the seed, having the seed forces the test cases to share configurations.

As the next step, we perform a post-mortem analysis to further reduce the number of configurations by eliminating the configurations that do not contribute to the coverage of t-way combinations for the test case (line 5). This step is needed only for those covering array generators, such as ACTS, that do not automatically eliminate non-contributing configurations in the seed.

The reduction is performed as follows: We iterate over all the configurations included in the newly computed covering array. For each configuration, we compute all the t-way option setting combinations present in the configuration. If there is at least one combination which is not covered by any other configurations in the covering array, we keep the configuration. Otherwise, we filter out the configuration, thus reduce the number of configurations needed.

We then merge the covering array, ca_{τ}, created for each test case with the system-wide covering array ca (line 6). Finally, after processing all the test cases, we output the computed t-way test case-aware covering array (line 8).

In this thesis, we use test case-aware covering arrays are to generate a test suite for testing at each iteration of our iterative testing process. At each iteration, we first identify t-way option setting combination-test case pairs that are at the risk of being masked. We then compute a t-way test case-aware covering array containing all such combinations. Finally, the newly computed test case-aware covering array is scheduled to be tested in the subsequent iteration.

Chapter 5

A Feedback Driven Adaptive Combinatorial Testing Approach

In this thesis, 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 define a novel interaction coverage criterion, called *tested t-way interaction coverage.* Under tested t-way interaction coverage criterion, 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, then no masking effects caused by option-related failures have occured and 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 can not determine an option-related cause, then we have not detected an option-related masking effect and must assume that all option setting combinations present in the configuration have been tested with that test case.

Note that we, in this work, are concerned with masking effects that are caused by option related failures. As a matter a fact, the way we remove these masking effects is by testing additional configurations. If we have a non-option-related failure, testing more configurations may not make the failure to disappear. Consequently, at the current state of the this work, when we have a non-option-related failure in a configuration, we simply assume that all the combinations present in the configuration have been tested (although this may not be the case).

In the rest of this thesis, we refer to each valid t-way option setting combination-test case pair as a *t-pair*. Furthermore, we define a masked t-pair as a t-pair (i.e., a valid t-way option setting combination-test case pair) that is not considered to be covered by the tested t-way coverage criterion.

To realize the tested t-way interaction coverage criterion in practice, we developed a feedback driven adaptive combinatorial testing process. This automated process takes as input a system, its configuration space model, and a set of test cases and iteratively attempts to achieve complete coverage under our criterion.

5.1 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 of the required 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-pairs. Return to step 2.

We now discuss each step of the process in detail. Following this section we present an equivalent, but more detailed algorithmic view of the process.

5.2 Step 1: Generating covering sets

At each iteration of our process, we compute a set of t-pairs 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.

To do this we maintain a separate configuration sub-model for each test case. This sub-model, in addition to inheriting all inter-option constraints that must be enforced globally, includes the test case-specific constraints. Furthermore, all t-way combinations of option settings considered to be already covered by the test case are expressed as a seed in the model (i.e., all the currently covered interactions).

We then feed the configuration sub-models to our test case-aware covering array generator discussed in Chapter 4. The result is a t-way test case-aware covering array to be tested.

5.3 Step 2: Execute test cases

Once we have t-way test case-aware covering array computed for the current iteration, we execute the test cases in the required configurations and record their pass/fail result.

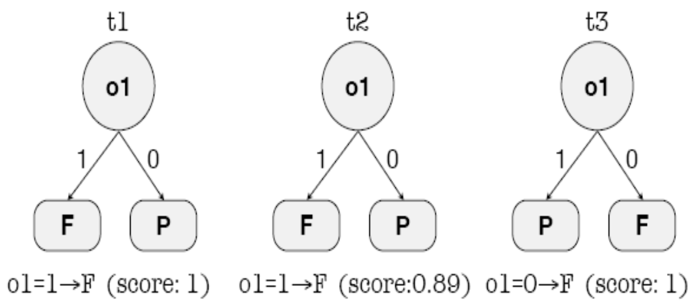
One optional step that we include in this thesis is that we treat the process of building the system as a special test, called a build test. This test must, of course, run before other traditional runtime tests unless a properly-configured, compiled system is available. For the build test, a passing 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 tests.

Figure 2a, as an example, depicts the test results obtained at the end of the first iteration of the process in a hypothetical scenario (the build test is omitted for clarity). In this scenario, t is 3 and the processes started for a configuration model that has 4 configuration options (o1, o2, o3, and o4). Each option takes a boolean value (0 or 1).

The system is tested with 3 test cases (t1, t2, and t3). Furthermore, initially, there were no known system-wide inter-option constraints and no test case-specific constraints. Consequently, the process automatically created a traditional 3-way covering array and executed all the test cases in all the configurations selected. Literals P and F indicate a test success or a test failure, respectively.

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 2: Classification model created for an example scenario

5.4 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.

To identify likely failure inducing option setting combinations, we feed the test results obtained to a classification tree algorithm. 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, feeding the test result data from Figure 2a to a classification tree algorithm could generate three classification models, one for each test case, such as those shown in Figure 2b. 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 a single leaf node indicating test case failure, but there could be more such leaf nodes 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 are highly correlated with the manifestation of failures. 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 computed by combining two standard metrics: precision (P), recall (R). For a given rule *R*, F1-measure is defined as follows:

$$\text{recall} = \frac{\#\text{ of correctly predicted failures by R}}{\text{total }\#\text{ of failures}}$$

$$\text{precision} = \frac{\#\text{ of correctly predicted failures by R}}{\text{total }\#\text{ of predicted failures by R}}$$

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

The F1-measure ranges between 0 and 1, inclusive. The higher the value, the better the rule is in predicting failures. Figure 2 shows the F1-measures for the rules obtained in our running example. For this thesis, an interaction is considered likely to be failure inducing, iff, the corresponding rule score is greater than a predetermined value, called *cutoff*. Any failures that are not explained by a significant failure cause are considered to be non-option-related.

As an example, consider the test t2 in Figure 2a. 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 a non-option-related failure.

5.5 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.

5.6 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 a test case-aware covering array 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 case-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 case-specific models for runtime tests incorporate any failure inducing interactions stemming from build test failures. Both these last two steps help the process avoid known failure causes in subsequent test iterations.

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

.

As an example, Figure 3a-b depict the configuration models created for the test cases in our running example after the analysis step. We are only showing the test constraints and seeds (the base model can be inferred from Figure 2a). The constraints were obtained by complementing the automatically identified failure inducing option setting combinations given in Figure 2b. The seeds were created by combining all the configurations in which the test cases ran successfully. Furthermore, Figure 3c illustrates the 3-way test case-aware covering array automatically scheduled for testing in the subsequent iteration given the configuration models presented in Figure 3a-b. Note that all the masking effects caused by a particular setting of o1 are removed in the newly computed covering array.

We furthermore examine each test case-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 the subsequent iteration.

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

Finally, we remove from further consideration, all test cases for which complete coverage has been achieved. If complete coverage has been achieved for each and every test (i.e., test case-aware covering array to be executed in the next iteration is empty), the process exits. Otherwise, we now generate the new covering set and return to step 2.

5.7 Algorithm

Given the previous discussion our basic algorithm should now be easy to follow. Algorithm 2 describes the main *adaptiveCA* routine. This routine defines and uses 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 one for each test case, 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 case-specific information in the respective data structures. All changes made on the subscripted variables are assumed to be reflected on the original variables.

Agorithm 2- adaptiveCA

```
Input t: Covering array strength
Input cfgMdl: Configuration Model
 1. $fMatrix \leftarrow$ empty
 2. $knownCauses \leftarrow$ empty
 3. while true
 4.   $currentCA \leftarrow$ computeTestAwareCA($t$, $cfgMdl$)
 5.   break if $currentCA$ is empty
 6.   $fMatrix \leftarrow$ executeCA($currentCA$, $fMatrix$)
 7.   foreach test $\tau$ do
 8.     $knownCauses_\tau \leftarrow$ identifyCauses($fMatrix_\tau$, $knownCauses_\tau$)
 9.     $cfgMdl_\tau \leftarrow$ updateCfgMdl($fMatrix_\tau$, $knownCauses_\tau$)
10.   end foreach
11. end while
```

The *adaptiveCA* routine loops until no test case is scheduled for testing. It takes an initial configuration model *cfgMdl*, and a strength *t*, as input. It first initializes the *fMatrix* and *knownCauses* data structures. At line 4, it creates an initial t-way test case-aware covering array for the configuration model provided. At line 6, all selected configuration-test case pairs are tested and the results are returned. These results are then analyzed at lines 7-10.

For each test case, we first identify the likely failure inducing interactions by using a classification tree algorithm (line 8). To compute the classification model, we create appropriate data files containing all the configurations tested so far, in which the test 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 5.1). 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 5.1) are also implemented at this step.

Next, we update the test case-specific configuration model (line 9). To do this we first, populate the list of the constraints for the test case with the newly identified test case-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 sub-spaces. 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 newly updated configuration model is then used to generate the test case-aware covering array to be executed in the next iteration. The iterations end when a complete coverage under our tested t-way coverage criterion is obtained for each and every test case, indicated by the current covering array to be tested being empty (line 5).

Chapter 6

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 compiler collection. Both systems are publicly-available.

6.1 Experimental Setup

For these experiments, we used the ACTS tool [6] 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 [7]. 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 Chapter 5).

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.

6.2 Study 1: MySQL Experiments

MySQL is an open-source, multi-threaded, SQL database management system (DBMS) [8]. 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.

6.2.1 Study Setup

We created an initial configuration model for MySQL containing 23 options (15 compile-time and 8 runtime). The number of settings varied among configuration options. We had 18 options with 2 levels of settings, 3 options with 3, 1 option with 4, and another option with 5 levels of settings (Table 6). 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 has its own oracle. The oracles categorize each test run into three classes: *passed*, *failed*, or *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 cluster 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.

Table 6: The configuration model of MySQL used in the study. ct and rt stand for compile-time and runtime, respectively

option	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
innodb_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
binlog-format	{row, statement, mixed}	rt
lb	{skip-log-bin,log-bin}	rt

6.2.2 Evaluation Framework

To precisely evaluate the proposed approach, we would need to know the number of valid interaction 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 have a definitive cause, and for which we know what the masking effects induced by the cause 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 option setting combination-test pairs that, we know for certain, never get tested with the test cases because of build failures or test skips. If a configuration fails to build, then we know none of the test cases gets 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 may 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.

6.2.3 Data and Analysis

We first ran the process with t = 2. Table 7 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 three columns present the number of unique errors, the unique test-errors 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 tests), analysis time (i.e., the time spent to identify likely failure inducing options and compute

Table 7: 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	223651	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

test schedules), and the total time, respectively. All the time measurements are given in minutes. Furthermore, all 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 a traditional 2-way testing, we observed that 55% (525149 out of 961795) of all valid 2-way combination-test 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 iterations 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 ∨ readline != with-readline) were scheduled to be tested in the next iteration. Testing them in the third iteration further 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 tests 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 cause was that some of the test skips were not deterministic. Here, some test cases skipped for reasons not related to the configuration. We believe 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 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 masking 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 would have been reduced by 60%, 11%, and 45%, respectively. This is important when the testing 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 8: Traditional t-way covering array-based testing

t	cfgs	test runs	lines covered	branches covered	errs	test-errs 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

Comparison with Higher Strength Traditional Covering 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 8 summarizes the performance on the traditional testing approaches.

We wanted to know how much higher strength covering arrays help in removing masking effects. Table 9 presents the numbers of 2-way combination-test 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 9 shows 1 minus the proportion of masked pairs obtained by our approach to masked pairs with the traditional approaches. For instance, our approach have 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.

Table 9: 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%

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 (133636 masked pairs for our approach vs. 133977 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 iteration 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 pairs. Table 10 depicts our results. It turned out that 44% of 15, 144, 193 valid 3-way combination-test 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 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 as the cutoff value, 96% of the constraints would have been correctly identified.

By the 4th iteration our approach reduced the number of 3-way combination-test pairs by 58% relative to the 1st 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 pairs masked by 6% (Table 11).

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. In particular, had we simply stopped after the first iteration as most current approaches do, significant numbers of interactions would have remained silently untested.

Table 10: Proposed approach with t = 3

	cfgs	test	lines	branches		test-	3-	testing	analysis	total
iteration	tested	runs	covered	covered	errs	err	masked	time	time	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

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.

Table 11: 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%

6.3 Study 2: GCC experiments

An important observation we made in the previous study is that much of the improvements came from the tests in which multiple different interactions caused different failures. In this study we explore this scenario with a some-what artificial test scenario. For this study we use gcc as our subject application.

Table 12: 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}

6.3.1 Study setup

We first studied the command line options of gcc and identified a small, but quite problematic configuration subspace. Table 12 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 up to a certain stage and then gcc exits. Syntax checks the code for syntax errors, but does not do anything beyond that. The preprocess option stops the compilation right after the preprocessing step; no compilation proper is performed. The assemble option 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 platform (as is the case in our experiments), gcc quits right away with an error message.

We then created a single test and its oracle. The test 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 [13]. 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 214 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*.

6.3.2 Data and Analysis

We ran our process with t = 2. Table 13 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 on the structural code coverage measurements up until the 10th 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 extent possible.

Table 13: 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

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), the chance of improvements 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, because of 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 tradition t-way covering array-based approaches. Table 14 presents the results. Our approach with t = 2 and t = 3 revealed all 14 failures as well as the golden configuration, whereas the traditional testing revealed at most 6 (including the 5-way covering array). None of the traditional arrays created for the study revealed the golden configuration.

We then computed the number or 1-, 2-, and 3-way option 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 removed all the masking effects and exercised all the t-way combinations that could be exercised. The 5-way covering array, 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.

Table 14: Comparing the proposed approach to traditional t-way testing in gcc experiment

approach	cfgs tested	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

6.4 Threads to Validity

We have identified several threads 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 thesis we do not examine that scenario. However, we believe that the scenarios studied in this thesis are realistic, since long times to defect fixes is common.

Chapter 7

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 feedback driven combinatorial testing approach. Instead of statistically 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 iteration 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 until 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 open source software system, MYSQL and GCC as subject applications. We observed that the proposed approach always significantly reduced the number of t-way combination-test pairs masked compared to traditional t-way covering arrays. The number of pairs was reduced by 66% when t = 2, and by 58% when t = 3. In both cases, only 19% of all valid intended combination-test pairs were 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 pairs masked compared with higher strength traditional covering arrays. The proposed approach with t = 2 reduced the number of 2-way combination-test pairs by 26% compared to using traditional 3-way testing. When t = 3 the approach reduced the 3-way combination-test pairs masked by only 6% compared to using larger 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.

As a summary, the results of our experiments strongly suggest that masking effects do exist and that our approach provides a promising and effcient way to work around them.

Chapter 8

Future Work

We think that this 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.

References

[1] D. Richard Kuhn, Raghu N. Kacker and Yu Lei, “Practical Combinatorial Testing,” *NIST Special Publication 800-142,* 2010.

[2] D. M. Kohen, S. R. Dalal, M. L. Fredman and G.C. Patton, “The AETG System: an approach to testing based on combinatorial design,” *IEEE Transaction on Software Engineering,* 23(7):437-44, 1997

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

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

[5] L. Breiman, J. Freidman, R. Olshen, and C. Stone, “Classification and Regression Trees,” *Wadsworth*, 1984.

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

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

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

[9] S. Fouche, 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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 (ICSM)*, pages 255-264, October 2007.

[15] R. Bryce, A. Rajan, M. P. E. Heimdahl, “Interaction Testing in Model Based Development: Effect on Model Coverage*,” IEEE, 13th Asia Pacific Software Engineering Conference (APSEC'06)* pp. 259-268 .

[16] K. Burr and W. Young, “Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage,” *International Conference on Software Testing,Analysis, and Review (STAR), San Diego, CA, October, 1998.*

[17] S. R. Dalal, C. L. Mallows, “Factor-covering Designs for Testing Software,” *Technometrics*, v. 40, pages 234-243, 1998.

[18] M. Grindal, J. Offutt, S. F. Andler, “Combination Testing Strategies: a Survey *Software Testing,” Verification, and Reliability*, vol. 15, pp. 167-199, 2005.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing," *Software Testing, Verification, and Reliability.*

[20] X. Yuan, M. B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", ASE '07: *Proceedings of the 22nd IEEE/ACM Intl. Conf. Automated Software Engineering*, pp. 405-408, November 2007.

[21] L.Tjen-Sien, L. Wei-Yin, S. Yu-Shan, "A comparision of Prediction Accuracy, Complexity, and Training Time of Thirty-three old and New Classification Algorithms," June 19 1997

[22] J. Ross Quinlan. C4.5 Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.

[23] C. Nie, and B.Xu "Using computational search to generate 2-way covering array," *State Key Laboratory for Novel Software Technology Nanjing University*, *Nanjing 210093, China* April 28, 2009.

[24] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios." *In Proceedings of the 24th Pacific Northwest Software Quality Conference*, 2006.

[25] C. Nie, B. Xu, L. Shi, And Z. Wang, "A new heuristic for test suite generation for pair-wise testing." *In Proceedings of the International Conference on Software Engineering and Knowledge Engineering,(SEKE'06)*, 517–521, 2006.

[26] M. Cohen, P. Gibbons, W. Mugridge, C. Colbourn, And J. Collofello, "Variable strength interaction testing of components," *In Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 413–418, 2003.

[27] A. Hartman, and L. Raskin, "Problems and Algorithms for Covering Arrays," *Discrete Math.,* vol. 284, pp. 149–156, 2004.

[28] C.J. Colbourn, "Combinatorial Aspects of Covering Arrays," *Le Matematiche (Catania)*, vol. 58, pp. 121–167, 2004.

[29] 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.

[30] 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.

[31] 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.

[32] D. Terzopoulos, A.Witkin, and M. Kass "Constraints on deformable models: Recovering 3D shape and nonrigid motion." *AI*, 36(1):91–123, 1988

[33] A. Hartman, "Software and hardware testing using combinatorial covering suites," *In M. C. Golumbic and I. B.-A. Hartman, editors, Graph Theory, Combinatorics and Algorithms, volume 34 of Operations Research/Computer Science Interfaces Series*, pages 237-266, Springer US, 2005.

[34] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," *In Proceedings of the 2007 international symposium on Software testing and analysis, ISSTA '07*, pages 129-139, New York, NY, USA, 2007.

[35] G. Mats, O. Jeff, and M. Jonas, "Handling constraints in the input space when using combination strategies for software testing," *Technical Report HS-IKI -TR-06-001, University of SkAuvde, School of Humanities and Informatics*, 2006.

[36] V. V. Kuliamin and A. A. Petukhov "A Survey of Methods for Constructing Covering Arrays" *Journal Programming and Computing Software,* vol. 37, May 2011.

[37] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," *In Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 49-59,Washington, DC, USA, 2004.

[38] G. Sherwood, “Effective testing of factor combinations,” *In Proceedings of the 3rd International Conference on Software Testing, Analysis, and Review(STAR94)*, 1994.

[39] S. Maity, AND A. Nayak, “Improved test generation algorithms for pair-wise testing,” *In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE’05). IEEE Computer Society, Los Alamitos, CA,* pages 235–244, 2005.

[40] N. Kobayashi, T. Tsuchiya, And T. Kikuno, “A new method for constructing pair-wise covering designs for software testing,” *Inform. Process. Lett. 81,* 2, pages 85–91, 2002.

[41] Ghazi, S.A. Ahmed, M. “Pair-wise test coverage using genetic algorithms.” *In Proceedings of the Congress on Evolutionary Computation (CEC’03).* IEEE *Computer Society, Los Alamitos, CA*, 1420–1424, 2003.

[42] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. Engineering of Computer-Based Systems (ECBS '07). 14th Annual IEEE International Conference and Workshops on the, pages 549-556, March 2007.

[43] 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.

[44] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Comput. Surv.*, 43:11:1-11:29, February 2011.

[45] R. C. Bryce and C. J. Colbourn, “One-test-at-a-time heuristic search for interaction test suites,” *In Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07*, pages 1082-1089, New York, NY, USA, 2007.