Test-aware Combinatorial Interaction Testing

Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul 34956, Turkey
cyilmaz@sabanciuniv.edu

ABSTRACT

Combinatorial interaction testing (CIT) approaches systematically sample a given configuration space and select a set of configurations, in which each valid t-way option setting combination appears at least once. A battery of test cases are then executed in the selected configurations. Existing CIT approaches, however, do not provide a systematic way of handling test-specific inter-option constraints. Improper handling of such constraints, on the other hand, causes masking effects, which in turn causes testers to develop false confidence in their test processes, believing them have tested certain option setting combinations, when they in fact have not. In this work, to avoid the harmful consequences of masking effects caused by improper handling of test-specific constraints, we compute *t-way test-aware covering arrays*. A t-way test-aware covering array is 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. We furthermore present a set of empirical studies conducted by using two widely-used highly-configurable software systems as our subject applications, demonstrating that test-specific constraints are likely to occur in practice and the proposed approach is a promising and effective way of handling them.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation

1. INTRODUCTION

General-purpose, one-size-fits-all software solutions are generally not acceptable in many application domains. For example, web servers (e.g., Apache), databases (e.g., MySQL), and application servers (e.g., Tomcat) are required to be customizable to adapt to particular run-time contexts and application scenarios. One way to support software customization is to provide configuration options through which the behavior of the system can be controlled.

While having a configurable system promotes customization, it creates a combinatorial configuration space, which may need extensive QA to validate. Since the size of a configuration space grows exponentially in the number of configuration options, exhaustively testing all configurations is generally not feasible. Therefore, a fundamental question is: *Which of the option setting combinations (i.e., configurations) should be tested?*

One solution approach, called combinatorial interaction testing (CIT), systematically samples the configuration space and tests only the selected configurations [2,5,8,14,21]. CIT methods take as input a configuration model that defines the valid configuration space for the software under test. This model typically includes a set of configuration options, each of which takes a value from a small number of discrete settings, and a set of system-wide constraints among configuration options. Given the model, these methods compute a t-way covering array – a set of configurations, in which each valid combination of option settings for every combination of t options appears at least once [5]. The system is then tested by running its test suite in all the configurations selected.

Covering arrays were initially proposed for testing input combinations of programs [5] and later adapted for testing software configuration spaces. In this work, we argue that there is a quite important distinction between using covering arrays for input combination testing and using them for configuration testing, which necessitates the development of dedicated approaches.

In input combination testing, each combination included in the covering array typically represents a concrete input to a test case and the software is tested with all of the input combinations selected. On the other hand, in configuration testing, each combination in the covering array represents a configuration, which barely passes as a test case. For adequate testing, a battery of *external test cases* need to be executed in all of the configurations selected.

External test cases may have assumptions about the underlying configurations; not all test cases may run in all configurations. In a study, we observed that 1% (378 test cases) and 46% (337 test cases) of all the test cases examined for Apache (a web server) and MySQL (a database management system) were configuration-dependent, i.e., had test-specific inter-option constraints. When the constraint of a test case

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

(a)

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

(b)

Figure 1: A traditonal 3-way covering array vs. a 3-way test-aware covering array.

is not met by a configuration, the test case simply *skips* the configuration, i.e., refuses to run in the configuration.

Existing covering array-based testing approaches do not provide a systematic way of handling test-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. However, when this assumption does not hold, existing approaches greatly suffer from masking effects – test skips that prevent some option-related behavior from being tested [9]. Masking effects in turn cause testers to develop false confidence in their test processes, believing them to have tested certain option setting combinations, when they in fact have not.

In this work, to avoid the harmful consequences of masking effects caused by improper handling of test-specific constraints, we propose three approaches to compute *t-way test-aware covering arrays*. The covering arrays we compute are not just a set of configurations as is the case in existing approaches, but a set of configurations, each of which is associated with a set of test cases. Two of the proposed approaches are geared towards minimizing the number of configurations included in test-aware covering arrays, whereas the third approach aims to minimize the number of test runs. To evaluate the proposed approach, we conducted a set of feasibility studies by using two widely-used highly-configurable software systems as our subject applications. In these studies, we demonstrate that test-specific constraints are likely to occur in practice and not taking them into account causes masking effects. Furthermore, the results of these studies suggest that the proposed approach is a promising and efficient way of handling test-specific constraints.

2. THE APPROACH

Figure 1a illustrates masking effects in a hypothetical test scenario. In this scenario, the system under test has 4 configuration options (*o1*, *o2*, *o3*, and *o4*), each of which takes a boolean value (0 or 1). The test suite contains 3 test cases (*t1*, *t2*, and *t3*). There is no system-wide constraint. However, the first two test cases have a test-specific constraint. Test case *t1* runs only in configurations in which $o1 = 0$ and test case *t2* runs only in configurations in which $o1 = 1$. When the constraint of a test case is not satisfied by a configuration, the test case skips the configuration (i.e., refuses to run in the configuration). Test case *t3*, on the other hand, has no test-specific constraint. There are 20 valid 3-way option setting combinations to cover for *t1* and *t2*, and 32 combinations for *t3*. The system is tested by using a traditional 3-way covering array. Since traditional covering array-based testing approaches do not provide a systematic way of taking test-specific constraints into account, all the test cases are executed in all of the configurations selected. Literals P and S indicates a test success or a test skip, respectively.

Consider test case $t1$. Since $t1$ skipped the first 4 configurations, the 3-way option setting combinations for options *o2*, *o3*, and *o4* that appear in the first 4 configurations were actually not tested by $t1$. As these 4 combinations appear nowhere else in the covering array, $t1$ never had a chance to test them. Following the same reasoning, test case *t2* never had a chance to test the 4 valid 3-way option setting combinations which happened to be present only in the last 4 configurations. As a result, 8 out of 72 (11%) valid 3-pairs were masked due to the test skips. A t-way option setting combination-test case pair is referred to as a *t-pair* in the remainder the paper.

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 others. 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-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-specific constraint of *t1* on *t3* would prevent *t3* from testing any combinations in which $o1 = 1$.

To prevent harmful consequences of masking effects caused by improper handling of test-specific constraints, we define a *t-way test-aware covering array*. In this approach, we take as input a configuration 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-specific inter-option constraints which are to be enforced per test case basis. Given a configuration model and a value of t, a t-way test-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.

Figure 1b presents as an example a 3-way test-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-pairs get to be tested.

Next, we present three algorithms to compute test-aware covering arrays. The first two algorithms are geared towards minimizing the number of configurations, whereas the third algorithm is geared towards minimizing the number of test runs required.

2.1 Algorithm 1: Maintaining a separate configuration submodel for each test case

The roots of this work stem from one of our earlier works [9], in which we introduced a feedback driven adaptive combinatorial testing process. In that work, to realize the proposed process in practice, we developed an approach to generate test-aware covering arrays. However, since test-aware covering arrays were not the main focus of the work, very little details about the approach were provided. We here provide further details.

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) 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-aware covering array for the entire test suite.

We in this work use a well-known tool, called ACTS [15], 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.

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.

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.

Algorithm 1 Computes a t-way test-aware covering array by maintaining a separate configuration submodel for each test case

Input t: Covering array strength
Input *ConfigModel*: System-wide configuration model

1: $ca \leftarrow empty$
2: **for** each test τ **do**
3: $\quad seed_\tau \leftarrow computeSeed(ConfigModel_\tau, ca)$
4: $\quad ca_\tau \leftarrow computeCA(t, ConfigModel_\tau, seed_\tau)$
5: $\quad ca_\tau \leftarrow reduce(ca_\tau)$
6: $\quad ca \leftarrow ca \cup ca_\tau$
7: **end for**
8: **return** ca

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 configuration, we keep the configuration. Otherwise, we filter out the configuration, thus reduce the number of configurations.

We then merge the covering array, ca_τ, created for the 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-aware covering array (line 8).

One downside of this approach is that the optimization is carried out per test case basis. While the problem is being solved for a test case, the coverage requirements of the remaining test cases waiting to be processed are not taken into account. This leads to loss of opportunity for further reducing the number of configurations.

2.2 Algorithm 2: Maintaining a single system-wide configuration model

In this section, to alleviate the shortcomings of the previous approach, we propose a greedy approach that maintains a global view of the test-specific constraints.

At each iteration, we select a configuration which covers the maximum number of t-pairs (i.e., t-way option setting combination-test case pairs) that have not been covered by the previous iterations (i.e., previous configurations). The iteration ends when there is no valid t-pair left uncovered.

Algorithm 2 depicts the high-level view of the approach. We maintain a pool that keeps track of the valid t-pairs yet to be covered. As the pairs are covered, they are removed from the pool. The pool initially contains all valid t-pairs (line 2). At each iteration, we pick the best row that covers the maximum number pairs currently present in the pool (line 4). A row in this context refers to a configuration

Algorithm 2 Computes a t-way test-aware covering array by maintaining a single system-wide configuration model.

Input t: Covering array strength
Input *ConfigModel*: System-wide configuration model

1: $ca \leftarrow empty$
2: $tpairs \leftarrow computeTPairsToCover(t, ConfigModel)$
3: **while** $tpairs$ is not empty **do**
4: $row \leftarrow pickBestRow(ConfigModel, tpairs)$
5: $tpairsCovered \leftarrow getTPairsCovered(ConfigModel, row)$
6: $tpairs \leftarrow tpairs - tpairsCovered$
7: $ca \leftarrow ca \cup row$
8: **end while**
9: **return** ca

together with a set of test-cases that are scheduled to be executed in the configuration.

Once a row is selected, we compute the set of t-pairs covered by the row (line 5) and remove these pairs from the pool (line 6). We then append the selected row to our test-aware covering array (line 7). Finally, when all the required pairs are covered (i.e., when the pool is empty), we output the computed test-aware covering array (line 9).

An integral part of the approach is computing the "best" row at each iteration. In this work, as a proof of concept, we implement this functionality using Answer Set Programming (ASP). ASP [16,19] is a declarative programming paradigm, which represents a computational problem as a "program" whose models, called "answer set", correspond to the solutions. ASP solvers are then used to find the answer sets for the program.

Algorithm 2 depicts our ASP encoding for an example scenario. This encoding computes the best row at a given iteration during the creation of a 2-way test-aware covering array for a simple configuration model. The configuration model contains three options (*o1*, *o2*, and *o3*). Option *o1* and *o2* take a binary value (i.e., 0 and 1), whereas *o3* takes 0, 1, or 2. There is no system-wide constraint. The system is tested using two test cases (*t1* and *t2*). Test case *t1* skips all the configurations in which $o1 = 1$. Test case *t2* has no constraint.

We now explain the encoding in a nutshell with no intention to introduce ASP. For more details about ASP, the interested reader may refer to an introduction [10] or a dedicated book [1].

In the configuration to be selected (*cfg*), each and every option must have exactly one valid setting:

```
1 {cfg(Opt,Val) : setting(Opt,Val)} 1 :- opt(Opt).
```

Configuration *cfg* is a valid configuration for test case T, if it is not an invalid configuration for the test case:

```
validCfg(T) :- test(T), not invalidCfg(T).
```

All the configurations in which $o1 = 1$ are invalid for test case *t1* (*t1* does not run in such configurations):

```
invalidCfg(t1) :- cfg(o1, 1).
```

Whereas the following line, as an example, indicates that 2-way option setting combination ($o1 = 0$, $o2 = 1$) is a valid combination that needs to be covered for test case *t1*.

```
tpair(o1, 0, o2, 1, t1).
```

```
% test cases
test(t1;t2).

% configuration options and their settings
opt(o1;o2;o3).
setting(o1;o2,0;1).setting(o3,0;1;2).

% configuration to be selected
1 {cfg(Opt,Val) : setting(Opt,Val)} 1 :- opt(Opt).

% the definition of a valid config. for a test T
validCfg(T) :- test(T), not invalidCfg(T).

% test constraints:
% an example test constraint: t1 skips when o1=1
invalidCfg(t1) :- cfg(o1, 1).
% ...

% pairs to cover:
% an example t-pair to cover
tpair(o1, 0, o2, 1, t1).
%...

% the definition of a covered pair
covered(O1,V1,O2,V2,T) :- cfg(O1,V1), cfg(O2,V2),
                          validCfg(T),
                          tpair(O1,V1,O2,V2,T).

% the optimization criteria
#maximize {covered(O1,V1,O2,V2,T)}.
```

Figure 2: ASP encoding for computing the "best" row at a given iteration.

Note that facts about invalid configurations and t-pairs to cover are configuration model specific and can be populated by an external driver, such as the one depicted in Algorithm 2.

A 2-way option setting combination ($O1 = V1$, $O2 = V2$), where *O1* and *O2* are options and *V1* and *V2* are valid settings, is considered to be covered for a test case T in configuration *cfg*, if 1) $O1 = V1$ and $O2 = V2$ in *cfg*, 2) *cfg* is a valid configuration for T, and 3) the combination needs to be covered for T:

```
covered(O1,V1,O2,V2,T) :- cfg(O1,V1), cfg(O2,V2),
                          validCfg(T),
                          tpair(O1,V1,O2,V2,T).
```

Finally, the following directive ensures that we pick the configuration that covers the maximum number of 2-pairs previously uncovered:

```
#maximize {covered(O1,V1,O2,V2,T)}.
```

For the simplicity of the discussion, we discussed our solution approach over an ASP encoding provided for a simple configuration model. Adapting the encoding to more complex configuration models and/or to compute higher strength covering arrays (i.e., $t > 2$) is straightforward. For example, in our experiments (Section 3), we developed a tool that automatically generated the ASP encoding for a given configuration model and a value of t.

2.3 Algorithm 3: Minimizing number of test runs

An interesting observation is that there is a trade-off between minimizing the number of configurations and minimizing the number of test runs. An attempt to minimize one count often results in increasing the other count. This trade-off is especially important when the cost of configuring the system and the cost of running the test cases are different.

The reason behind the trade-off is a simple one. Forcing test cases to share configurations, thus reducing the number of configurations, may cause a test case to execute in a number of configurations to cover a certain set of t-way option setting combinations, which could have been covered by less number of configurations, thus reducing the number of runs of the test case, if the test case was not forced to share configurations. On the other hand, not sharing configurations across the test cases increases the number of configurations.

The proposed approaches presented in Section 2.1 and 2.2 are geared towards minimizing the number of configurations. We now introduce an approach geared towards minimizing the number of test runs.

In this approach, we slightly modify Algorithm 1 such that, instead of creating a seed for each test case, which has all the configurations that have been included in the covering array so far (thus forcing the test case to share these configurations when possible), we create an empty seed. That is, line 3 in Algorithm 1 is replaced with $seed_\tau \leftarrow empty$. The rest of the algorithm stays the same.

In effect, we provide each test case with freedom to select its own configurations in order to minimize the number of times it is executed. Test cases are scheduled to be executed in the configurations they select.

3. EXPERIMENTS

We conducted a series of studies to 1) demonstrate that test-specific constraints are likely to occur in practice, 2) demonstrate that traditional covering array-based testing approaches suffer from masking effects caused by improper handling of test-specific constraints, and 3) evaluate the proposed approach.

In the studies, we used two widely-used highly-configurable software systems as our subject applications: Apache v2.3.11-beta and MySQL v5.1. Apache is an HTTP server. MySQL is a database management system. Both systems enjoy a large developer community that actively updates and tests them.

All the experiments were performed on a dual Intel Xeon processor machine with 2GB of RAM, running the CentOS 5.2 operating system.

3.1 Study 1: Test-specific constraints

In our first study, to demonstrate that configuration-dependent test cases are likely to occur in practice, we examined the test suites that came with the source code distribution of our subject applications.

Each test case in the test suites has its own test oracle which determines whether each test case execution "passed", "failed", or was "skipped". Successful test cases simply emit pass. Failed test cases emit fail. A test case returns skipped when it determines that it cannot run in a given configuration.

Table 1: Distribution of Apache test cases over clusters, each which is identified by a unique constraint.

cluster idx	# of tests	# of options	cluster idx	# of tests	# of options
1	172	3	10	5	3
2	74	1	11	4	1
3	26	1	12	3	2
4	22	1	13	2	2
5	21	1	14	2	2
6	16	1	15	2	1
7	11	1	16	2	1
8	8	1	17	1	1
9	7	2			

To detect and identify the test-specific inter-option constraints, we studied the test oracles and, as needed, manually investigated the test cases, read the user manuals, and conducted experiments. Out of 3789 and 738 test cases studied for Apache and MySQL, respectively, we identified 378 (1%) Apache test cases and 337 (46%) MySQL test cases that run only in certain configurations, i.e., that have some test-specific constraints.

We then determined the actual test-specific constraints for these configuration-dependent test cases. It turned out that the constraints involved a total of 13 and 9 unique configuration options for Apache and MySQL, respectively.

One interesting remark is we observed that the test cases formed clusters with respect to their constraints. That is, we had clusters of test cases sharing exactly the same constraints. We identified 17 and 29 such clusters for Apache and MySQL, respectively.

Table 1, as an example, provides some statistics about the distribution of 378 configuration-dependent Apache test cases over 17 clusters, each of which is identified with a unique constraint. The columns in the table depict the cluster indices, the number of test cases included in each cluster, and the number of unique configuration options involved in the constraints associated with the clusters. The first row, for instance, indicates that 172 test cases share exactly the same constraint involving 3 configuration options.

We observed a similar trend in MySQL test cases. Among the 29 clusters identified, the largest cluster had 86 test cases and the smallest cluster had 1 test case.

This observation is particularly important towards improving the scalability of test-aware covering array generators. Instead of handling each and very test case, test cases can be divided into clusters and then the covering array can be created by using one sample test case taken from each cluster. This could considerable reduce the number of t-pairs that need to be dealt with at runtime. Once a test-aware covering array is created for the sample test cases, each sample test can then be replaced with all the test cases in the respective cluster. We followed this approach to create the test-aware covering arrays discussed in Section 3.

In this study, we demonstrated that configuration-dependent test cases are likely to occur in practice. We furthermore learned that such test cases tend to form clusters with respect to their constraints.

3.2 Study 2: Masking effects

In this study we evaluate the harmful effects of not being able to properly handle test-specific constraints. For that purpose, we mimic the way that the traditional covering ar-

Table 2: Configuration model for Apache.

option	setting
case-filter	{–disable-case-filter,–enable-case-filter}
ssl	{–disable-ssl,–enable-ssl}
dav	{–disable-dav,–enable-dav}
auth-digest	{–disable-auth-digest,–enable-auth-digest}
echo	{–disable-echo,–enable-echo}
rewrite	{–disable-rewrite,–enable-rewrite}
case-filter-in	{–disable-case-filter-in,–enable-case-filter-in}
bucketeer	{–enable-bucketeer,–disable-bucketeer}
info	{–enable-info,–disable-info}
headers	{–enable-headers,–disable-headers}
vhost-alias	{–enable-vhost-alias,–disable-vhost-alias}
cgi	{–enable-cgi,–disable-cgi}
imagemap	{–enable-imagemap,–disable-imagemap}
proxy-http	{–enable-proxy-http,–disable-proxy-http}
proxy	{–enable-proxy,–disable-proxy}
System-wide constraint	
proxy-http = –enable-proxy-http → proxy=–enable-proxy	

Table 3: Configuration model for MySQL.

option	settings
asm	{NULL,enable-assembler}
linfile	{NULL,enable-local-infile}
bt	{NULL,with-big-tables}
ec	{NULL,with-extra-charsets=complex, with-extra-charsets=all}
innodb	{with-innodb,without-innodb}
libedit	{with-libedit,without-libedit}
ndbcluster	{NULL,with-ndbcluster}
readline	{with-readline,without-readline}
ssl	{NULL,with-yassl}
ase	{NULL,with-archive-storage-engine}
bse	{NULL,with-blackhole-storage-engine}
fse	{NULL,with-federated-storage-engine}
sql_mode	{strict_all_tables, traditional, ansi}
log-format	{row, statement, mixed}
lb	{skip-log-bin,log-bin}
System-wide constraint	
ssl = NULL ∧ (libedit=with-libedit → readline = without-readline)	

rays are typically used in practice. In particular, we create a configuration model for our subject applications, create traditional covering arrays, schedule all the test cases to run in all the configurations selected, and quantify the consequences of masking effects caused by test skips.

To carry out the study, we first created a configuration model for each subject application. All the configuration options that cause test skips were included in these models.

Table 2 and 3 depict the configuration model of Apache and MySQL used in the study. Both configuration models have 15 options and one system-wide constraint. System-wide constraints are included in the configuration models to avoid invalid configurations; configurations that violate these constraints either do not get built or cause runtime exceptions. All the configuration options in the configuration model of Apache are binary, whereas the configuration model of MySQL has 12 options with binary settings and 3 options with 3 levels of settings.

We then created traditional covering arrays with varying strengths for our configuration models by using ACTS. For each value of t and the choice of subject application, we created 10 covering arrays and schedule all the test cases of interest to execute in all the configurations selected. In this study, we consider only the configuration-dependent test cases identified in Study 1. The rest of the test cases are ignored.

Table 4 presents the average number of configurations and the number of test runs we obtained (column 3 and 4). A test case scheduled to be executed in a configuration is counted as one test run.

To quantify the harmful consequences of masking effects caused by improper handling of test-specific constraints, we use a metric, called t-masked [9]. This metric counts the number of unique t-pairs that are untested because of test skips. To compute the value of t-masked, for each and every test case, we first count the number of t-way option setting combinations that are present only in the configurations that the test skips. We then add these counts up across all the test cases.

The last two columns of Table 4 present the 2-masked and 3-masked values computed for the traditional covering arrays used in the study. We observed that the traditional covering arrays greatly suffered from masking effects caused by test skips. For example, when $t = 2$, 44403 (32.68%) of 138246 valid 2-pairs, and, when $t = 3$ 247452 (22.21%) of 1114099 valid 3-pairs never had a chance to be tested (i.e., masked) in 2-way and 3-way covering arrays created for Apache.

In the presence of masking effects, one way to reduce the number of t-pairs being masked is to use higher strength traditional covering arrays, i.e., use a t'-way covering array to obtain t-way coverage, where $t' > t$. For instance, if we use a 3-way covering array to prevent 1- or 2-way option setting combinations from being masked, since we expect that each 1- and 2-way combination will appear multiple times in different 3-way combinations, the masking may be avoided.

As Table 4 indicates, we observed that although using higher strength covering arrays reduced masking effects, they did not solve the problem entirely. For example, using the traditional 3-way covering arrays reduced the value of 2-masked from 44403 to 9824 compared to using the 2-way covering arrays, and using the 4-way covering arrays further reduced it to 240 for Apache.

Note that using a sufficiently large value of t' would certainly prevent all t-pairs (where $t' > t$) from being masked. However, it would do so at the cost of increased number of configurations and test runs. For instance, our 4-way covering arrays created for Apache have 160% more configurations and test runs compared to the 2-way covering arrays.

From this study, we learned that traditional t-way covering arrays greatly suffer from masking affects caused by improper handling of test-specific constraints. Furthermore, using higher strength traditional covering arrays tend to reduce masking effects, but they do so at the cost of increased amount of resources required for testing.

3.3 Study 3: Test-aware covering arrays

In this study, we populate our configuration models used in Study 2 with the test-specific constraints identified in Study 1. We then use the proposed approaches to create test-aware covering arrays.

Table 5 presents the results we obtained. In this table, MPT refers to the approach in which we create one configuration submodel for each test case (Section 2.1) and MPS refers to the approach in which we use one system-wide con-

Table 4: Traditional covering arrays.

sut	t	configs	test runs	2-masked	3-masked
Apache	2	10	3780	44403 (32.68%)	n/a
Apache	3	26	9828	9824 (7.11%)	247452 (22.21%)
Apache	4	73	27594	240 (0.02%)	17633 (1.58%)
MySQL	2	12	4044	43933 (31.16%)	n/a
MySQL	3	38	12806	6816 (4.84%)	233178 (19.32%)
MySQL	4	101	34037	412 (0.29%)	34382 (2.85%)

Table 5: Test-aware covering arrays.

sut	t	approach	configs	test runs
Apache	2	MPT	54.5	4127.8
Apache	2	MPS	26.1	4044.8
Apache	2	MTR	127.2	3780.0
Apache	3	MPT	123.2	10702.9
Apache	3	MPS	68.2	9671.0
Apache	3	MTR	324.6	8901.1
MySQL	2	MPT	78.8	5015.2
MySQL	2	MPS	43.3	5722.3
MySQL	2	MTR	238.9	3891.1
MySQL	3	MPT	238.8	14062.7
MySQL	3	MPS	132.7	15990.0
MySQL	3	MTR	694.3	10815.4

figuration model (Section 2.2). In the realization of the MPS approach, to find the best row at each iteration, we executed the respective ASP encoding for at most 3 minutes; the best solution found (could be the optimal solution) in 3 minutes is used. Both MPT and MPS aim to minimize the number of configurations. MTR, on the other hand, refers to the approach in which we aim to minimize the number of test runs (Section 2.3). For each approach and the value of t, we created 10 test-aware covering arrays. The table reports the average sizes of the covering arrays obtained.

One observation is that, for the configuration models used in the study, the t-way test-aware covering arrays had more configurations, but not necessarily more test runs compared to the traditional t-way covering arrays. For instance, the 3-way test-aware covering arrays created by the MPS approach for Apache required more configurations (68.2 vs. 26), but less test runs (9671 vs. 9828) compared to the traditional 3-way covering array. This is typically to be expected. Handling test-specific constraints typically increases the number of configurations needed, as the t-way combinations being masked for the test cases need to be covered in additional configurations. However, this does not necessarily increase the number of test runs, as the test cases are scheduled to be executed only in configurations contributing to their coverage.

We also observed that having a global view of coverage requirements for the test cases (i.e., the MPS approach) was better at reducing the number of configurations compared to having a partial view of the test requirements (i.e., the MPT approach). When $t = 2$, MPS, compared to MPT, provided covering arrays with 52% less configurations for Apache and with 45% less configurations for MySQL. Similarly, when $t = 3$, MPS, compared to MPT, reduced the number of configurations by 45% and 44% for Apache and MySQL, respectively.

Furthermore, the test-aware covering arrays prevented all masking effects and the ones created by the MPS approach did so at a fraction of the cost compared to using higher strength traditional covering arrays, which were not even able to remove all the masking effects. For example, when $t = 2$, the test-aware covering arrays created by MPS, compared to the traditional 4-way covering arrays, reduced the number of configurations by 64% and 57%, and the number of test runs by 85% and 83% for Apache and MySQL, respectively. When $t = 3$, compared to the traditional 4-way covering arrays, the number of configurations and the number of test runs were reduced by 7% and 65% for Apache. For MySQL, although we observed 31% increase in the number of configurations (keeping in mind that the 4-way traditional covering arrays suffered from 34382 3-pairs being masked), the number of test runs was reduced by 53%.

Finally, using the MTR approach, as expected, reduced the number of test runs while increasing the number of configurations needed. When $t = 2$, MTR, compared to MPS, reduced the number of test runs by 7% and 32% while increasing the number of configurations by 387% and 452% for Apache and MySQL, respectively. Similarly, when $t = 3$, the number of test runs was reduced by 8% and 32%, while the number of configurations was increased by 376% and 423%. Clearly, if the cost of configuring the system is negligible, the MTR approach will be preferable.

4. RELATED WORK

The problem of generating covering arrays is NP-hard [18]. In the literature, four main types methods have been proposed to generate covering arrays: greedy methods [5, 7], heuristic search-based methods [4, 12], mathematical methods [13], and random search-based methods [20]. Nie et al. provide a comprehensive survey of these methods [18]. The proposed approaches presented in this work fall into the category of greedy methods. However, compared to the previous greedy approaches, the properties of the covering arrays we compute are different.

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 [6]. Mats et al. propose various techniques for efficient handling of constraints [17]. Bryce et al. introduce "soft constraints"' to mark option setting combinations that are permitted, but undesirable to be included in a covering array [3]. These approaches are mainly concerned with system-wide inter-option constraints. We, on the other hand, provide various approaches to handle test-specific constraints.

Seeding mechanisms in the generation of covering arrays have been used to guarantee the testing of certain configurations [5, 7, 11]. However, in this work, we use a seeding mechanism to force test runs to share configurations as much as possible, which potentially reduces the number of configurations required.

5. CONCLUSION

In this work, to avoid the harmful consequences of masking effects caused by improper handling of test-specific constraints, we proposed three approaches to compute t-way test-aware covering arrays. The covering arrays we compute

are not just a set of configurations as is the case in existing approaches, but a set of configurations, each of which is associated with a set of test cases. Two of the proposed approaches are geared towards minimizing the number of configurations included in test-aware covering arrays, whereas the third approach aims to minimize the number of test runs.

To evaluate the proposed approach, we conducted a set of feasibility studies by using two widely-used software systems, Apache and MySQL, as our subject applications.

In these studies, we first demonstrated that configuration-dependent test cases, thus test-specific constraints, are likely to occur in practice. We observed that 1% (378 test cases) and 46% (337 test cases) of all the test cases examined for Apache and MySQL were configuration-dependent. We then demonstrated that traditional covering arrays greatly suffer from masking effects caused by improper handling of test-specific constraints. In the experiments, 32% (21%) of all the required 2-pairs (3-pairs) were masked on average, i.e., never had a chance to get tested, when traditional 2-way (3-way) covering arrays were used. Although using higher strength traditional covering arrays reduced the masking effects, they did so at the cost of increased amount of resources required for testing. On the other hand, our test-aware covering arrays prevented all masking effects from occurring and they did so at a fraction of the cost compared to using higher strength traditional covering arrays. Finally, we demonstrated that there is a trade-off between minimizing the number of configurations and minimizing the number of test runs; an attempt to minimize one count often results in increasing the other count.

We believe that this line of research is interesting with some potential practical impact. As the next step, we plan to develop a generic test-aware covering array generator which takes various types of costs into account, e.g., cost of reconfiguration and cost of running tests.

6. ACKNOWLEDGMENTS

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (FP7-PEOPLE-IRG-2008), and by the Scientific and Technological Research Council of Turkey (109E182).

7. REFERENCES

[1] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, England, 2003.

[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. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[4] 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. ACM.

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

[6] 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. ACM.

[7] J. Czerwonka. In *Proc. of the 24th Pacific Northwest Software Quality Conference*.

[8] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Int'l Conf. on Software Engineering*, pages 285–294, 1999.

[9] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. *To appear in the Proc. of Int'l Syposium on Software Testing and Analysis (ISSTA)*, July 2011.

[10] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Tutorial Lectures*, volume 5689 of *LNCS*, pages 40–110. Springer, 2009.

[11] S. Fouché, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 557–560, 2007.

[12] S. Ghazi and M. Ahmed. Pair-wise test coverage using genetic algorithms. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 2, pages 1420 – 1424 Vol.2, dec. 2003.

[13] 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.

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

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

[16] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, 1999.

[17] 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 SkÃűvde, School of Humanities and Informatics, 2006.

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

[19] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[20] 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. IEEE Computer Society.

[21] 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.