Cost-Aware Combinatorial Interaction Testing

Gulsen Demiroz and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, Turkey
Email: {gulsend,cyilmaz}@sabanciuniv.edu

***Abstract*—The configuration spaces of modern software systems are often too large to test exhaustively. Combinatorial interaction testing approaches (CIT), such as covering arrays, systematically sample the configuration space and test only the selected configurations. Traditional t-way covering arrays aim to cover all t-way combinations of option settings in a minimum number of configurations. By doing so, they assume that the testing cost of a configuration is the same for all configurations. In this work, we however argue that, in practice, the actual testing cost may differ from one configuration to another and that accounting for these differences can improve the cost-effectiveness of covering arrays. In this work, we first introduce a novel combinatorial object, called a *cost-aware covering array*. A t-way cost-aware covering array is a t-way covering array that minimizes a given cost function. We then provide a framework for defining the cost function. Finally, we present an algorithm to compute cost-aware covering arrays for a simple, yet important scenario, and empirically evaluate the cost-effectiveness of the proposed approach. The results of our empirical studies suggest that cost-aware covering arrays, depending on the configuration space model used, can greatly reduce the actual cost of testing compared to traditional covering arrays.**

I. Introduction

The configuration spaces of configurable software systems are often too large to test exhaustively. The number of possible configurations is often far beyond the available resources to test the entire configuration space in a timely manner, e.g., for regression testing.

Combinatorial interaction testing (CIT) approaches take as input a configuration space model. The model includes a set of configuration options, each of which can take on a small number of option settings. As not all configurations may be valid, the model can also include some system-wide inter-option constraints. In the context of this work, an inter-option constraint is a constraint that implicitly or explicitly invalidates some combinations of option settings. In effect, the configuration space model implicitly defines a set of valid ways the software under test can be configured.

CIT approaches systematically sample the valid configuration space and test only the selected configurations. The sampling is carried out by computing a combinatorial object, called a *covering array*. Given a configuration space model, a t-way covering array is a set of configurations, in which each possible combination of option settings for every combination of t options appears at least once [6].

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. The results of many empirical studies strongly suggest that a majority of option-related failures in practice are caused by the interactions among only a small number of configuration options and that traditional t-way covering arrays, where t is much smaller than the number of options, are an effective and efficient way of revealing such failures [6], [2], [10], [9].

Existing approaches construct a t-way covering array in such a way that all valid t-way combinations of option settings are covered by using a minimum number of configurations. By doing so, these approaches implicitly assume a simple cost model where the cost of configuring the system under test is the same for all configurations.

In this work we, however, argue that this cost model is not always valid in practice. First, we observe that the configuration cost often varies from one configuration to other. For example, in a study conducted on MySQL – a widely-used and highly-configurable database management system, we observed that the cost of configuring the MySQL Community Server (a core component of the system) with its default configuration took about 6 minutes on average[1]. On the other hand, configuring the system with NDB cluster storage support – a feature that enables clustering of in-memory databases, and with embedded server support – a feature that makes it possible to run a full-featured MySQL server inside a client application, took about 9 minutes, as these features needed to be compiled into the system. Therefore, in a covering array, reducing the number of configurations that include these features, without adversely affecting the coverage of option setting combinations, can significantly reduce the amount of time required for testing. However, existing approaches do not take actual testing costs into account when computing covering arrays.

Second, we observe that highly configurable systems often have reusable components which, once configured, can be used in other configurations with no or very little additional

[1]Performed on an 8-core Intel(R) Xeon(R) CPU 2.53GHz machine with 32 GB of RAM, running CentOS 6.2 operating system.

A traditional 2-way covering array

o1	o2	o3	o4	o5	o6	o7
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	1	0	1
1	0	0	0	0	1	1
1	0	1	1	1	1	1
1	1	0	1	0	1	0
1	1	1	0	0	0	0

(a)

2-way covering array for options {o1,o2,o3}

- **Options:**
  {o1,o2,o3}: {0,1}
- **Constraints:**
  <empty>

o1	o2	o3
0	0	0
0	1	1
1	0	1
1	1	0

(b)

A 2-way cost-aware covering array

- **Options:**
  {o1,o2,o3,o4,o5,o6,o7}: {0,1}
- **Constraints:**
  (o1=0 ∧ o2=0 ∧ o3=0)∨
  (o1=0 ∧ o2=1 ∧ o3=1)∨
  (o1=1 ∧ o2=0 ∧ o3=1)∨
  (o1=1 ∧ o2=1 ∧ o3=0)

o1	**o2**	**o3**	o4	o5	o6	o7
0	**0**	**0**	0	0	0	0
0	**0**	**0**	1	1	0	0
0	**0**	**0**	1	1	1	1
0	**0**	**0**	0	1	0	1
0	**1**	**1**	0	0	1	1
1	**0**	**1**	1	1	1	1
1	**0**	**1**	1	0	1	0
1	**1**	**0**	0	0	0	0

(c)

Figure 1. (a) A traditional 2-way covering array. (b,c) Illustrates our algorithm where (b) shows 2-way covering array for only compile-time options and (c) shows 2-way cost-aware covering array.

cost. One simple example is the presence of compile-time and runtime configuration options.

Compile-time options need to be set before the system can be built. The system is then configured as a part of the build process. Therefore, changing the setting of a compile-time option requires a partial or a full rebuild of the system. On the other hand, given a build of the system, runtime options are set when the system is running and the system is configured on the fly. Note that a build of the system is a reusable component. Once the system is built for a given combination of compile-time option settings, the same build can be used with different runtime configurations without any additional cost; as long as the settings of compile-time options stay the same, the same binaries can be reused. However, runtime configurations are not reusable. Even for the same build (i.e., the same compile-time configuration) they need to be reconfigured every time the program is executed, unless the program state is saved for future use.

Figure 1(a) and 1(c) illustrate the effect of reusable components on testing cost in a hypothetical scenario. In this scenario, we have 7 configuration options $o1, \ldots, o7$, each of which can take on a binary value (i.e., 0 or 1). The first 3 options $o1$, $o2$, and $o3$ are compile-time options, whereas the remaining options $o4$, $o5$, $o6$, and $o7$ are runtime options. There are no system-wide inter-option constraints; all option setting combinations are valid. Furthermore, the system is to be tested with a 2-way covering array. Two covering arrays are created for comparison.

The 2-way covering array presented in Figure 1(a) includes 8 unique combinations of compile-time option settings, requiring to build the system 8 times. On the other hand, the 2-way covering array presented in Figure 1(c) requires to build system only 4 times, as it includes 4 unique compile-time configurations. For example, once the system is built for $o1$=0, $o2$=0, and $o3$=0, the same binaries are reused without any additional cost for 3 more configurations included in the covering array. Assuming that the runtime configuration cost is negligible compared to the compile-time configuration cost and that the compile-time configuration cost is the same for all configurations, the 2-way covering array in Figure 1(c) tests all 2-way option setting combinations at half of the cost compared to the 2-way covering array in Figure 1(a).

To improve the cost-effectiveness of CIT approaches, we in this work introduce a novel combinatorial object, called a *cost-aware covering array*. Given a traditional configuration space model augmented with a cost function and a value of t, a t-way cost-aware covering array is a t-way covering array that minimizes the cost function. We, furthermore, provide an algorithm to compute cost-aware covering arrays for a simple, yet frequently-faced scenario in practice. The results of our empirical studies suggest that cost-aware covering arrays, depending on the configuration space model used, can greatly reduce the actual cost of testing compared to traditional covering arrays.

The remainder of the paper is organized as follows: Section II discusses related work; Section III introduces cost-aware covering arrays; Section IV presents an algorithm to compute cost-aware covering array for a particular cost model; Section V describes the empirical studies; Section VI

presents concluding remarks and possible directions for future work.

II. Related Work

In this section we provide background information on traditional covering arrays and discuss related work.

Traditional CIT approaches take as input a configuration space model $M=<O,V,Q>$. The model includes a set of configuration options $O=\{o_1,o_2,\ldots,o_n\}$, their possible values $V=\{V_1,V_2,\ldots,V_n\}$, and some system-wide inter-option constraints Q (if any). Each configuration option o_i ($1 \leq i \leq n$) takes a value from a finite set of $|V_i|$ distinct values $V_i = \{v_{i1}, v_{i2}, \ldots, v_{i|V_i|}\}$.

Definition 1. *Given a configuration space model $M=<O,V,Q>$, a t-tuple $\phi_t=\{<o_{i_1},v_{j_1}>, <o_{i_2},v_{j_2}>, \ldots, <o_{i_t},v_{j_t}>\}$ is a set of option-value tuples for a combination of t distinct options, such that $1 \leq t \leq n$, $1 \leq i_1 < i_2 < \ldots < i_t \leq n$, and $v_{j_p} \in V_{i_p}$ for $p=1,2,\ldots,t$.*

Not all the t-tuples may be valid due to the constraints Q. Let $valid(\phi_t, Q)$ be a deterministic function such that $valid(\phi_t, Q)$ is true, if and only if, ϕ_t satisfies the constraint Q. Otherwise, $valid(\phi_t, Q)$ is false. The set of all *valid* t-tuples Φ_t under constraint Q is then defined as: $\Phi_t=\{\phi_t : valid(\phi_t, Q)\}$.

Definition 2. *Given a configuration space model $M=<O,V,Q>$, a valid configuration c is a valid n-tuple, i.e., $c \in \Phi_n$, where $n = |O|$.*

Definition 3. *Given a configuration space model $M=<O,V,Q>$, the valid configuration space C is the set of all valid configurations, i.e., $C=\{c : c \in \Phi_n\}$.*

Definition 4. *A t-way covering array $CA(t, M=<O,V,Q>)$ is a set of valid configurations in which each valid t-tuple appears at least once, i.e., $CA(t, M=<O,V,Q>)=\{c_1, c_2, \ldots, c_N\}$, such that $\forall\, \phi_t \in \Phi_t\, \exists\, c_i \supseteq \phi_t$, where $c_i \in C$ for $i=1,2,\ldots,N$.*

The problem of generating covering arrays is NP-hard [15]. Nie et al. classify the methods for generating covering arrays into 4 main categories [15]: random search-based methods [16], heuristic search-based methods [8], [4], [7], [11], [4], [17], greedy methods [6], [9], [5], [19], [18], [14], and mathematical methods [20], [13], [21], [12].

Random search-based methods employ a random selection without replacement strategy [16]. Valid configurations are randomly selected from the configuration space in an iterative fashion until all the required t-tuples have been covered by the configurations selected.

Heuristic search-based methods, on the other hand, employ heuristic search techniques, such as hill climbing [8], tabu search [4], and simulated annealing [7], or AI-based search techniques, such as genetic algorithms [11] and ant colony algorithms [17], to compute covering arrays. These methods typically maintain a set of configurations at any given time and iteratively apply a series of transformations to the set until the set constitutes a t-way covering array.

Greedy algorithms also operate in an iterative manner [6], [9], [5], [19], [18], [14]. At each iteration, among the sets of configurations examined as candidates, the one that covers the most previously uncovered t-tuples is included in the covering array and the newly covered t-tuples are then marked as covered. The iterations end when all the required t-tuples have been covered.

Mathematical methods for constructing covering arrays have also been studied [20], [13], [21]. Some mathematical methods are based on recursive construction methods, which build covering arrays for larger configuration space models (i.e., the ones with a larger number of configuration options) by using covering arrays built for smaller configuration space models (i.e., the ones with a smaller number of configurations) [20], [13]. Other mathematical methods leverage mathematical programming, such as integer programming, to compute covering arrays [21].

Our approach differs from existing covering array generators in that we compute a t-way covering array that minimizes a given cost function, rather than computing a covering array that minimizes the number of configurations required.

Furthermore, Bryce et al. introduce the concept of soft constraints to mark option setting combinations that are permitted, but undesirable to be included in a covering array [3]. Although soft constraints could be used to avoid costly combinations of options settings, thus to reduce testing cost, using soft constraints for this purpose can be considered to be an opportunistic approach. Our approach, on the other hand, takes the task of reducing the cost as an optimization criterion.

III. Cost-Aware Covering Arrays

In our approach we take as input a traditional configuration space model augmented with a cost function $cost(.)$. Given a covering array ca, $cost(ca)$ returns the expected cost of testing ca.

Definition 5. *Given a configuration space model $M=<O,V,Q,cost(.)>$ and a value of t, a t-way cost-aware covering array is a t-way covering array that minimizes the function $cost(.)$.*

Defining the cost function is not a trivial task. For example, the cost of a given covering array may not simply be the sum of the cost of the configurations included in the array, as some parts of a configured system can be reused by other configurations with no or little additional cost. Therefore, we present a framework for defining the cost function.

Definition 6. *Given a configuration space model* $M=<O,V,Q>$*, a* component class $X=\{o_{i_1}, o_{i_2}, \ldots, o_{i_k}\}$ *is a set of* k *distinct options, such that* $X \subseteq O$*.*

Definition 7. *Given a* component class $X=\{o_{i_1}, o_{i_2}, \ldots, o_{i_k}\}$*, a* component x *is a* k*-tuple of the form* $\{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots, <o_{i_k}, v_{j_k}>\}$ *for the configuration options included in* X*, where* $k=|X|$*.*

We assume that the set of configuration options O are divided into p ($1 \leq p \leq |O|$) component classes $X_1, X_2, \ldots, X_p$, such that $X_i \cap X_j = \emptyset$ for $i \neq j$ and $X_1 \cup \ldots \cup X_p = O$. Consequently, a given configuration c is composed of p components $x_1, x_2, \ldots, x_p$, such that x_i is a component of component class X_i for $i=1, \ldots, p$.

For example, in our running example depicted in Figure 1, we have two component classes: $X_1=\{o1, o2, o3\}$ and $X_2=\{o4, o5, o6, o7\}$. Component class X_1 includes all the compile-time options, whereas X_2 includes all the runtime options.

We distinguish between two types of component classes: reusable and non-reusable component classes.

Definition 8. *A reusable component class* X^r *is a component class whose components can be configured in isolation and, once configured, they can be reused in other configurations.*

Definition 9. *A non-reusable component class* X^{nr} *is a component class whose components need to be configured every time they are used.*

Going back to our running example, we observe that X_1 is a reusable component class, since, once the system is built for a given compile-time configuration, the resulting binaries can be reused in other configurations with different runtime configurations. On the other hand, X_2 is a non-reusable component class, since the runtime options need to be configured every time the system is executed.

To determine the cost of a given covering array, we assume two cost functions: $cc(.)$ and $lc(.)$. The function $cc(x)$ takes as input a component x (either a reusable or a non-reusable component) and returns the configuration cost of x. For example, assuming that the reusable component x represents a configuration for a library, $cc(x)$ is the cost of compiling the library with the given configuration. The function $lc(c)$, on the other hand, takes as input a configuration c and returns the cost of linking (i.e., gluing) together the components appearing in the configuration. For example, assuming that a configuration c is composed of reusable components x_1^r and x_2^r, each of which represents a library, $lc(c)$ is the cost of linking the two libraries after they are compiled, i.e., after the $cc(x_1^r)$ and $cc(x_2^r)$ costs are paid.

Definition 10. *The cost of a configuration* c*, which is composed of components* $x_1, x_2, \ldots, x_p$*, is defined as*

$$\left(\sum_{1 \leq i \leq p} cc(x_i)\right) + lc(c)$$

However, in the presence of reusable components, the cost of a given covering array is *not* the sum of the cost of the configurations included in the array.

Definition 11. *Given a covering array* $ca=\{c_1, c_2, \ldots, c_N\}$*, let* R_i *and* S_i *be the set of reusable and non-reusable components in a configuration* c_i*, respectively, where* $1 \leq i \leq N$*. The cost of the covering array* ca *is then defined as follows:*

$$cost(ca)=\sum_{x \in \bigcup_{1 \leq i \leq N} R_i} cc(x) + \sum_{1 \leq i \leq N} \left(lc(c_i) + \sum_{x \in S_i} cc(x)\right)$$

Furthermore, reusable components can form a hierarchy.

Definition 12. *A reusable composite component is a component, which is composed of reusable components and/or other reusable composite components.*

Reusable composite components are constructed by linking the components appearing in the composite, once these components are configured. Therefore, to account for composite components, the $lc(.)$ function should ensure that the linking cost of the same reusable composite components is paid only once.

IV. Computing Cost-Aware Covering Arrays for a Simple Cost Model

We conjecture that all the methods that have so far been used to compute traditional covering arrays, such as random search-based methods, heuristic search-based methods, greedy methods, and mathematical methods (Section II), can also be used to construct cost-aware covering arrays, all with their own pros and cons. In this work, however, we, as a proof of concept, present an algorithm to compute cost-aware covering arrays for a simple, yet important cost model.

In this cost model, the system under test has compile-time and runtime options. For a given configuration space model of the system, we define two components X^r and X^{nr}. X^r is a reusable component class, containing all the compile-time options in the model, whereas X^{nr} is a non-reusable component class, containing all the runtime options in the model. We assume that (1) the cost of linking compile-time and runtime configurations is negligible, i.e., $lc(c)=0$ for all c, (2) the compile-time configuration cost is the same for all compile-time configurations, i.e., $cc(x^r)=a$ for some constant a for all x^r, and (3) the runtime configuration cost of the system is negligible, i.e., $cc(x^{nr})=0$ for all x^{nr}.

Under this cost model, the cost of a covering array $ca=\{c_1, c_2, \ldots, c_N\}$ is

$$cost(ca)=a \times |\bigcup_{1 \leq i \leq N} R_i|,$$

where a is the constant cost of building the system, and R_i is the set of compile-time components appearing in configuration c_i ($1 \leq i \leq N$). In other words, under this model the optimization criterion is to minimize the number of times the system is built, while covering all t-tuples.

Although this cost model may seem to be overly constrained at a first glance, since our goal in this paper is to demonstrate the differences between the cost-effectiveness of traditional and cost-aware covering arrays, rather than to compute cost-aware covering arrays for any given cost function, we believe that the cost model employed serves well to its purpose.

Furthermore, based on our feasibility studies conducted on MySQL – a highly-configurable database management system, and Apache – a highly-configurable HTTP server, we argue that this simple cost model still has some practical importance. For example, we observed that (1) both subject applications have compile-time and runtime options, (2) runtime configuration cost for both subject applications is negligible, (3) the cost of linking runtime configurations with compile-time configurations is negligible. Although, for both subject applications, compile-time configuration costs vary from one configuration to another, since building these systems from scratch is costly, reducing the number of times they are built is still of practical value, e.g., building the entire software suite that comes with the source code distribution of our subject applications with the default configuration takes about 80 minutes for MySQL and 8 minutes for Apache, on average.

With all these in mind, Algorithm 1 presents our algorithm. In this algorithm, we use traditional covering array construction as a computational primitive. In particular, we assume a generator $\prod(t, M)$ that constructs a traditional t-way covering array for the configuration space model M.

Given a configuration space model M and a value of t, our algorithm operates as follows: (1) a traditional t-way covering array Ω is generated for only the compile-time options (line 1), (2) all the compile-time configurations included in the newly computed array are expressed as an inter-option constraint Q (line 3-5), (3) a traditional t-way covering array Ψ satisfying Q, is generated for all the configuration options (line 6). The output Ψ (line 7) is a t-way cost-aware covering array, minimizing the number of compile-time configurations, i.e., minimizing the number of times the system is required to be built.

The rationale behind this algorithm is a simple one. Step (1) selects a "minimal" set of compile-time configurations covering all t-way combinations of option settings for the compile-time options. Step (2), by expressing these compile-time configurations as constraints, ensures that step (3) computes a traditional t-way covering array around these configurations without introducing new compile-time configurations, "minimizing" the number of compile-time configurations required, thus the testing cost.

Algorithm 1 Computes a t-way cost-aware covering array

Input M=<$O, V, \emptyset$>: Configuration space model
Input t: Covering array strength
Let M' be the configuration space model for only the compile-time options in M

1: $\Omega \leftarrow \prod(t, M')$
2: $Q \leftarrow \emptyset$
3: **for each** $c = \{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots\}$ **in** Ω **do**
4: $\quad Q \leftarrow Q \vee \{o_{i_1} = v_{j_1} \wedge o_{i_2} = v_{j_2} \wedge \ldots\}$
5: **end for**
6: $\Psi \leftarrow \prod(t,\ M$=<$O, V, Q$>)
7: **return** Ψ

Figure 1(b) and (c) illustrate the algorithm in our running example introduced in Section I. First, a traditional 2-way covering array is generated for the 3 compile-time options $o1$, $o2$, and $o3$ (Figure 1b). The array has 4 compile-time configurations. Second, these configurations are expressed as a constraint so that no additional compile-time configurations can be selected (Figure 1c). Finally, a traditional 2-way covering array satisfying the constraint is generated for all the options. The resulting cost-aware covering array requires to build the system under test 4 times.

V. Experiments

To evaluate the proposed approach, we conducted a set of experiments.

A. Experimental Setup

To carry out the experiments, we first implemented our algorithm. In the implementation, we used a well-known and widely-used covering array generator: ACTS (v1.r9.3.2) [1].

We then determined a base configuration space model and varied the model in a systematic and controlled manner to obtain other models. For each configuration space model obtained, we computed a traditional t-way covering array and a t-way cost-aware covering array, and compared their cost-effectiveness, i.e., compared the number of builds required by these arrays.

All the experiments were performed on an 8-core Intel(R) Xeon(R) CPU 2.53GHz machine with 32 GB of RAM, running CentOS 6.2 operating system.

B. Independent Variables

In particular we experimented with 3 independent variables:

- m: The number of compile-time options in the configuration space model. We experimented with m=$5, 6, \ldots, 20$.
- m/n: The ratio of compile-time options to the total number of options in the configuration space model,

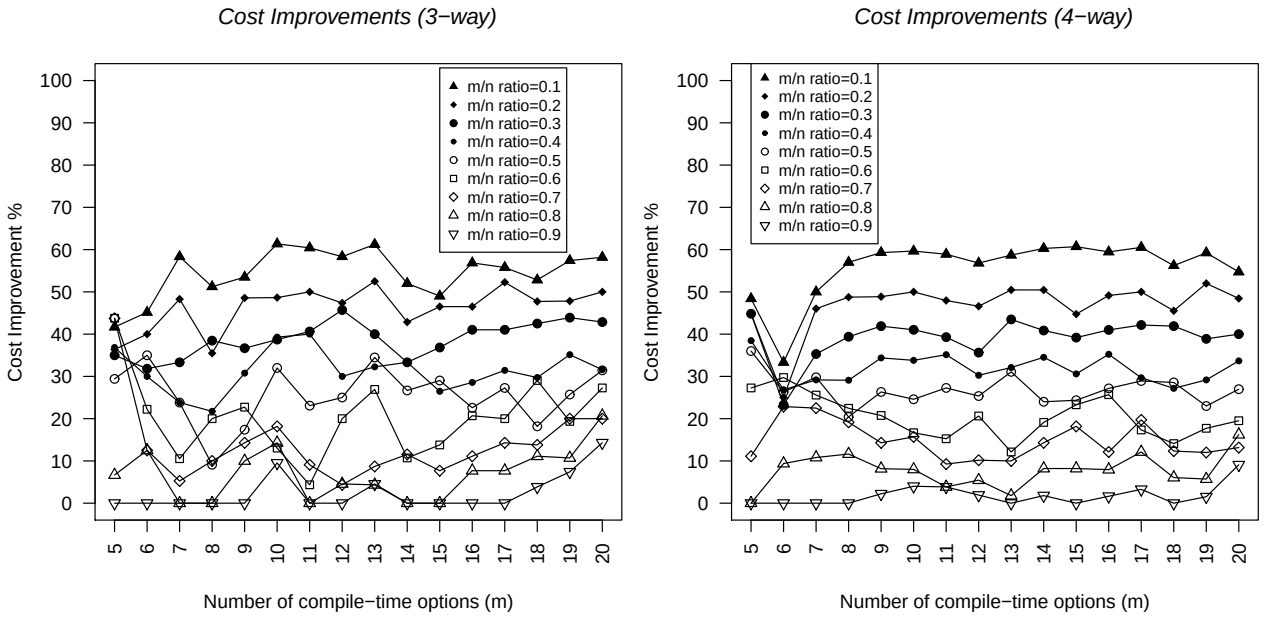

Figure 2. Cost Improvements in a) 3-way b) 4-way cost-aware covering arrays with different m.

where n is the total number of options and $n - m$ is the number of runtime options. We experimented with m/n=$0.1, 0.2, \ldots, 0.9$.

- t: The strength of the covering array. We experimented with t=$3, 4$.

In all the configuration space models, we, without losing the generality, used binary options only. Given m and m/n ratio, the respective configuration space model is obtained by adding binary runtime options to the model, such that the requested m/n ratio is attained. Furthermore, we opted not to experiment with t=2 because for the m and m/n values used in the experiments, the sizes of the covering arrays generated were similar to each other. This made it difficult to analyze the effect of our independent variables on the cost-effectiveness of cost-aware covering arrays.

C. Evaluation Framework

To evaluate the cost-effectiveness of cost-aware covering arrays and compare it to that of traditional covering arrays, we counted the number of unique compile-time configurations required by the arrays. That is, we counted the number of times the system is required to be built. Note that this is indeed the optimization criterion dictated by the cost model our algorithm is designed for (Section IV).

When creating the traditional covering arrays, we configured ACTS to create partially filled covering arrays. In a partially filled covering array, some option settings are left unset, indicating that, regardless of the actual settings used for these, as long as they are valid settings for the respective options, the array will still be a covering array. Once a partially filled traditional covering array was obtained, we followed a greedy approach to pick the best settings for the unset options so that the number of compile-time configurations is "minimized". Had we had ACTS to create fully filled covering arrays, the unset options would have been randomly set, which could have increased the number of compile-time configurations required. Therefore, the fully filled traditional covering arrays used in the comparisons represent the best case scenario for the partially filled covering arrays created by ACTS.

D. Data Analysis

Figure 2a-b present the results we obtained. In these figures, the horizontal axis denotes the values of m (i.e., the number of compile-time options) used in the experiments, whereas the vertical axis depicts the percentage of cost improvements (i.e., percentage of decrease in the number of compile-time configurations required) provided by cost-aware covering arrays over traditional covering arrays. Figure 2a is for t=3 and Figure 2b is for t=4.

We first observed that the cost-effectiveness of cost-aware covering arrays were better or the same (but never worse) compared to that of traditional covering arrays. More accurately, when t=3, the cost-effectiveness of cost-aware covering arrays were better than that of traditional covering arrays in 89% (128 out of 144) of the comparisons. In the remaining comparisons (i.e., 11% of the comparisons), the cost-effectiveness of the arrays were the same. When t=4,

Table I
3-WAY AND 4-WAY COST IMPROVEMENT (%) AVERAGES FOR DIFFERENT M/N RATIOS.

m/n ratio	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
3-way	54.58	46.31	38.86	31.31	25.63	20.27	14.03	6.89	2.48
4-way	55.83	46.80	39.25	31.83	26.88	20.45	14.80	7.72	1.83

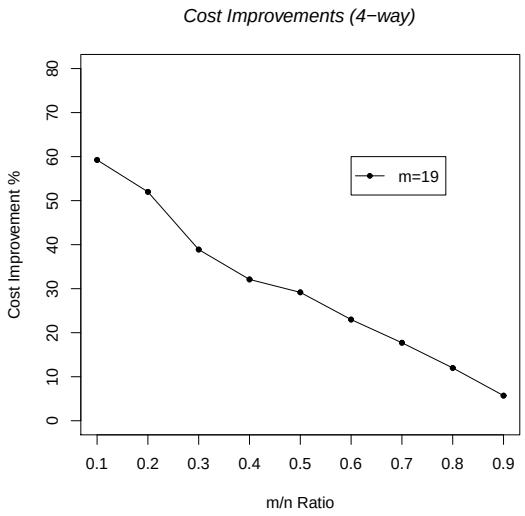

Figure 3. Cost Improvements in 4-way cost-aware covering arrays with different m/n ratio for $m = 19$.

the cost-effectiveness of cost-aware covering arrays were better in 94% and the same in 6% of the comparisons.

We then observed that actual cost improvements provided by cost-aware covering arrays varied depending on the m/n ratio used in the configuration space models. For a fixed m, as the m/n ratio increased, cost improvements tended to decrease. Table I presents the cost improvement percentages provided by cost-aware covering arrays. For example, when t=4 and m=19, the cost-aware covering arrays, compared to the traditional covering arrays, reduced the cost by 59.24%, 52%, 38.89%, 32.1%, 29.17%, 22.99%, 17.72%, 12%, 5.71% when m/n=$0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$, respectively (Figure 3). Clearly, when m/n=1, regardless of the value of m, as the configuration space model will include only compile-time options, there will be no difference between the cost-effectiveness of traditional and cost-aware covering arrays.

For the values of m and m/n used, when the m/n ratio was fixed, the cost improvements tended to be stable regardless of the value of m. On the other hand, when $m \leq t$, as the compile-time configurations will be tested significantly, there will be no difference between the cost-effectiveness of traditional and cost-aware covering arrays.

Furthermore, comparing 4-way and 3-way cost-aware covering arrays with traditional covering arrays, we observed that 4-way cost-aware covering arrays tended to provide slightly more cost improvements than 3-way cost-aware covering arrays; as t was increased from 3 to 4, the cost improvements over traditional covering arrays tended to increase (Table I). For example, when m/n=0.1, the average cost improvement provided by 3-way cost-aware covering arrays was 54.58%, whereas 4-way cost-aware covering arrays provided 55.83% cost improvement.

VI. CONCLUDING REMARKS

In this paper, we first introduced a novel combinatorial object, called a *cost-aware covering array*. Unlike traditional t-way covering arrays, which aim to minimize the number of configurations required to cover all valid t-tuples, t-way cost-aware covering arrays aim to cover all t-tuples in a set of configurations, which minimizes a given cost function. Given a set of configuration, the cost function computes the actual cost of testing. Furthermore, since computing the testing cost in configuration spaces is a nontrivial task, especially in the presence of reusable components, we provided a framework for defining the cost function. Finally, we presented an algorithm to compute cost-aware covering arrays for a particular cost scenario, and empirically evaluated the cost-effectiveness of cost-aware covering arrays.

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. One potential threat is that our algorithm was designed for a particular cost scenario. However, the cost scenario used in the paper, although simple, is of great practical importance.

Another possible threat to external validity concerns the representativeness of the configuration space models used in the experiments. Although we systematically varied the models and evaluated the cost-effectiveness of the proposed approach, i.e., a total of 288 different models were used in the experiments (16 values of m $\times$ 9 values of m/n $\times$ 2 values of t), these models are still one suite of models. A related issue is that the configuration space models used in the experiments did not contain any inter-option constraints. While these issues pose no theoretical problems (our algorithm can be modified to account for constraints), we need to apply our approach to more realistic configuration space models in future work.

Despite these limitations, we believe our study supports our basic hypotheses. We reached this conclusion by noting that our studies showed that: (1) in practice, the testing cost

may not be the same for all configurations, (2) accounting for the presence of reusable components, i.e., the components, which, once configured, are reused in other configurations, can reduce the testing cost, (3) minimizing the number of configurations as is the case in traditional covering arrays does not necessarily minimize the actual cost of testing, and (4) the cost-aware covering arrays were generally more cost-effective than the traditional covering arrays used in the experiments.

We believe that this line of research is novel and interesting, but much work remains to be done. We are therefore continuing to develop new approaches that overcome existing limitations and threats to external validity. In particular, we are developing tools and algorithms that are based on metaheuristic search techniques, such as simulated annealing, to compute cost-aware covering arrays for any given configuration space model and for any cost function.

REFERENCES

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

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

[3] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *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] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Reliab.*, 19:37–53, March 2009.

[6] 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.

[7] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[9] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 285–294, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] N. Kobayashi. *Design and evaluation of automatic test generation strategies for functional testing of software*. Osaka University, Osaka, Japan, 2002.

[14] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18:125–148, September 2008.

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

[16] 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.

[17] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 72–77, Washington, DC, USA, 2004. IEEE Computer Society.

[18] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1):109 –111, jan 2002.

[19] Y.-W. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431 – 437 vol.1, 2000.

[20] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[21] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 283–, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.