Journal of Statistical Theory and Practice (2020) 14:7
https://doi.org/10.1007/s42519-019-0075-2

ORIGINAL ARTICLE

Covering Arrays: Using Prior Information for Construction, Evaluation and to Facilitate Fault Localization

Ryan Lekivetz[1] · Joseph Morgan[1]

Published online: 2 December 2019

Abstract

Covering arrays are increasingly being used by test engineers to derive test cases to test complex engineered systems. This approach to testing is known as combinatorial testing and has proven to be a cost-efficient way to determine test cases that are highly effective at identifying faults in the system that are due to the combination of several inputs. However, when such faults are encountered and failures occur, the test engineer is tasked with determining the inputs and associated values that triggered the failures. This exercise typically involves examining a long list of potential causes and may even require performing follow-up tests to reduce the number of potential causes. This paper addresses this issue by considering the prior knowledge of the system under test that test engineers often have. We show how this knowledge can be used to evaluate and improve the effectiveness of a suite of test cases before any test cases are executed. Finally, we address the case where failures occur and show how this prior knowledge can aid in determining the inputs, and associated values, that triggered the failures. In addition, throughout the paper, we compare and contrast the use of covering arrays for testing complex engineered systems to the use of factorial experiments in traditional experimental design settings.

Keywords Covering arrays · Design of experiments · Fault localization · Prior information

Part of special issue guest edited by Pritam Ranjan and Min Yang—Algorithms, Analysis and Advanced Methodologies in the Design of Experiments.

✉ Ryan Lekivetz
ryan.lekivetz@jmp.com

Joseph Morgan
joseph.morgan@jmp.com

[1] JMP Division, SAS Institute, Cary, USA

1 Introduction

Testing complex engineered systems is a difficult undertaking where, for a system under test (SUT) and its corresponding input space, test engineers are tasked with constructing a set of test cases that can efficiently identify faults in the SUT. Test engineers are typically limited by tight budgetary constraints and, as a result, testing all possible combinations of the inputs is infeasible. As is the case in a traditional experimental design setting, focusing on only one input at a time is inefficient. Furthermore, when faults are due to the combination of settings for two or more inputs, testing one input at a time is likely to miss some combinations of settings and is also ineffective. In a statistical design of experiments setting, a fractional factorial design would be one way to address such a problem. While deriving test cases to test a complex engineered system can be thought of as being conceptually similar to developing an experimental design [7], there are some important differences. First, the SUT must be deterministic, thus ensuring reproducible testing results. This also implies that replicated runs add no additional information to a test suite, and need not be considered. Also, the measured response (outcome) of interest to the test engineer is binary, pass or fail, where a pass indicates that for a given test case, the actual behavior of the SUT corresponds to the expected behavior, while a fail indicates that actual behavior deviates from expected behavior. In addition, after testing a complex engineered system, when failures occur, the outcomes of the test cases are used by the test engineer to isolate the underlying cause of observed failures, not to estimate effects as would be done in a statistical design of experiments setting.

One of the approaches to testing that seeks to address the challenge of testing such complex engineered systems is combinatorial testing. Combinatorial testing currently implies a covering array as the underlying mathematical construct [19]. The columns of the covering array correspond to inputs of the SUT and rows correspond to test cases. Covering arrays ensure that all possible combinations of levels among t inputs appear in a test suite, where t is known as the strength of the covering array. Note that orthogonal arrays are covering arrays but, whereas every level combination involving t or fewer inputs appears equally often for orthogonal arrays, every level combination involving t or fewer inputs needs to only appear at least once for covering arrays. This less restrictive property means that covering arrays in general need much fewer runs compared to the corresponding orthogonal array for the same number of inputs. For strength 2 covering arrays with binary inputs, the problem of finding the covering array having the minimal number of runs has been solved [11, 13, 21]. Table 1 provides some examples comparing the run size for

Table 1 Comparison of optimal binary covering arrays (CA) to orthogonal arrays (OA) for k inputs

k	2–3	4	10	15	35	56	126	1000
CA	4	5	6	7	8	9	10	14
OA	4	8	12	16	36	60	128	1004

Cell values indicate the minimum run size required for specified value of k

strength 2 covering arrays (CA) versus orthogonal arrays (OA) for k binary inputs. Note that the strength of a covering array can be thought of as the projectivity of an orthogonal array [1].

The efficiency and effectiveness of the combinatorial testing approach derives from the economic run size and the coverage property of covering arrays. Consider a set of test cases derived from a covering array. If all the test cases result in a pass, then the coverage property allows a test engineer to assert that there are no faults due to combinations involving t or fewer inputs. However, when there are failures, the test engineer is faced with the task of fault localization—identifying the inputs and their level combinations that precipitated the failures. Fortunately, the test engineer typically has prior knowledge of the system, through inputs, or combinations of inputs that may have been problematic in the past or due to recent changes that may not have been rigorously tested.

While the goal of testing complex engineered systems is not to measure effects, for those familiar with factorial designs, the fundamental principles for factorial effects [22] are instructive when considering testing such systems. The effects in this case are not model effects, but rather failure-inducing combinations of inputs. We will review these fundamental principles, from the factorial effect standpoint, before discussing how they relate to combinatorial testing.

Effect hierarchy/combination hierarchy: (1) *Lower-order effects are more likely to be important than higher-order effects*. In an empirical study of software failures across a variety of domains by Kuhn et al. [16], nearly all faults were due to some combination of values for five or fewer inputs. When investigating a failure, a test engineer will start with the potential failure-inducing combinations involving the fewest number of inputs.

(2) *Effects of the same order are equally likely to be important.* Without prior knowledge of the system, a test engineer will treat all combinations of the same number of inputs as being equally likely to induce a failure. However, a test engineer may have knowledge of the system that allows the likelihood of some combinations of the same order to be more likely to induce a failure [18]. The use of this knowledge is shown in Sect. 4.

Effect sparsity/combination sparsity: *The number of relatively important effects will be small.* For combinatorial testing, sparsity of failure-inducing combinations is critical to attain useful information about failures induced by a test suite. If there are too many failure-inducing combinations, the SUT is perhaps not ready for combinatorial testing as almost every test results in a failure.

Effect heredity/combination heredity: *An interaction is significant only if at least one of the parent factors involving the interaction is significant.* As stated, the effect heredity principle does not apply to combinatorial testing. If there is a failure due to a certain combination of k input levels, then any higher-order combination containing that k input combination will also result in a failure. The failures can then only be attributed to the lower-order failure-inducing combination. Then, once a system has no single input that precipitates a failure, the notion of a significant parent factor is not appropriate. However, knowledge of the system is particularly useful in identifying inputs more likely to be involved in a fault. That is, higher-order combinations involving certain inputs may be more likely to induce a failure.

Viewed in this way, the appeal of covering arrays becomes more apparent. With effect hierarchy, a test engineer wants to focus their attention on lower-order combinations of inputs. The guarantee of covering all combinations of inputs involving *t* or fewer inputs is precisely what a test engineer wants in a test suite. The sparsity principle allows the test engineer to benefit from the run size economy of covering arrays and still have confidence that faults can be detected and identified. If we reframe heredity in a way that uses prior information about inputs more likely to be involved in a fault, it can benefit both the design of a test suite and our ability to do fault localization.

Section 2 provides examples of complex engineered systems. In Sect. 3, we present notation and discuss some preliminary details. Section 4 discusses quantifying prior knowledge through weights, while Sect. 5 uses these weights to evaluate the effectiveness and aid in the construction of covering arrays before testing begins. In Sect. 6, we show how fault localization can be aided by providing a ranking of the inputs and associated values of the set of potential causes that trigger failures. We conclude the paper with a discussion of future work in Sect. 7.

2 Motivation

Let us consider XGBoost, a widely used open-source gradient boosting software library [3] which is an example of a complex engineered system. Before running XGBoost, users often tweak several parameters to get the best possible performance from the underlying algorithms. There are several categories of parameters and, within each category, there are usually many parameters. For example, Tree Booster is one of four boosters that XGBoost offers (release 0.90), and there are twenty-two parameters to configure Tree Booster. An example of one of these parameters is learning rate, which ranges between 0 and 1 and is used to control overfitting. Another example is tree method, which allows the user to specify one of five tree construction algorithms. Since most of these twenty-two parameters are continuous values, the configuration space for the parameters of Tree Booster alone is infinitely large. A test engineer faced with validating XGBoost must therefore contend with several challenges. First, the engineer must ensure that the parameters that are used to configure XGBoost work as intended. Second, the underlying algorithms that constitute XGBoost must also work as intended.

As it turns out, most modern software systems present the dual validation challenge that XGBoost presents [20]. To further complicate matters, test engineers must also take into account the nature of failures that may occur when validating such systems. Failures may take many forms and the severity of a failure could range from catastrophic to benign. For example, in [14], the authors describe a traffic collision avoidance system (TCAS) that is mandated by civil aviation authorities around the world for commercial aircraft of a certain size. TCAS is an example of a complex engineered system where any failure could be catastrophic. On the other hand, in [15], the authors discuss a case where the complex engineered system is a browser. Failure for such systems could be benign or may even be costly but would not be catastrophic in the way that failure of a TCAS-like system would.

We will use an example from a commercially available software product (see Fig. 1) to illustrate the fault localization discussion in Sect. 6. The example is similar to XGBoost but is simpler in that there are only six user-configurable options. These options allow users to specify how they would like the application to present its output. This simple, but real, example will allow us to demonstrate how the ideas in this paper would help a test engineer to identify failure-inducing input combinations.

3 Notation and Preliminaries

Consider an array D with n rows and l columns. Let column i have s_i levels for $i = 1, \dots, l$. D is said to be a covering array of strength t if any subset of t columns has the property that all $\prod s_i$ level combinations occur at least once.

Covering arrays may be used to derive a suite of test cases for combinatorial testing. If we map the inputs of the SUT to the columns of a covering array, then the levels of the columns would map to the allowed values of the corresponding inputs, and the rows of the covering array would then be the test cases of the test suite. For continuous factors, one may need to partition the input into discrete levels. In software testing, this is referred to as *equivalence partitioning*. Since a strength t covering array ensures that all combinations of values for any t inputs are covered then a covering array-based combinatorial testing approach allows test engineers to be confident that faults due to t or fewer inputs will be discovered, which are the most likely to exist by the combination hierarchy principle. If instead, the test engineer generated a set of test cases of the same size by randomly selecting from the input space, then there is a nonzero probability that combinations for t or fewer inputs would be missed.

For many systems, devising a test suite is only the beginning. If a failure is discovered, the test engineer typically wants to know which combination of inputs and associated levels induced the failure. This is known as the fault localization

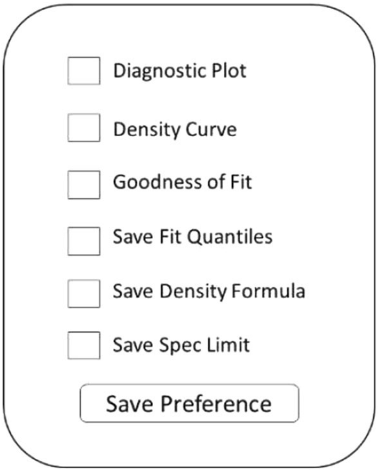

Fig. 1 Set of inputs for Example 3

problem [9]. To devise the list of potential causes, exact methods have been proposed to evaluate the outcomes of test suites [6]. In the simplest form, for combinations involving k inputs, the exact method starts with all input combinations involved in test cases that induce a failure, and removes input combinations that appear in the test suite for test cases in which a failure did not occur. The value of k investigated is typically the smallest value for which the set of potential causes is non-empty [16], which we adopt throughout this paper. However, it is not uncommon for a test engineer to be faced with a long list of potential causes, and so examining the list of potential causes can be an onerous task.

Before using a test suite, there are a number of considerations for fault localization that can be made, based on test cases in the test suite. For instance, let us assume that failures occur on a subset of test cases, while all remaining test cases result in passes. Define $\psi_k(\{m_1, \ldots, m_b\})$ as the set of potential causes due to k inputs if failures only occur in rows $m_1, \ldots, m_b$ of the test suite with all other tests passing. Define $C_{i_1,\ldots,i_k}(j_1, \ldots, j_k)$ as the combination of level j_1 for input i_1, level j_2 for input i_2, and so on. If $C_{i_1,\ldots,i_k}(j_1, \ldots, j_k)$ induces a failure, any row in the test suite that contains this combination will result in a failure. If there is only one failure ($b = 1$), a test engineer needs to examine the k-input combinations in $\psi_k(\{m_1\})$ and find the failure-inducing combination. When $b > 1$, there may be potential causes that appear on more than one row, while others may appear in one row only. If $b > 1$, and a combination that appears in only one row is in fact a failure-inducing combination, then the remaining failure(s) still need to be accounted for. For example, when $b = 2$, there may be a combination in $\psi_k(\{m_1, m_2\})$ that would singly account for the failures in m_1 and m_2, while the remaining combinations would belong to m_1 alone and m_2 alone. Under the assumption of sparsity, a single combination that can account for both failures is more likely to be the actual cause of the failures than different combinations.

Ideally, when a failure is due to a particular input combination, we would like to be able to directly identify the actual cause. Error-locating arrays [5] can provide this capability, but at a cost of additional runs in the test suite. For fault localization, we propose a metric based on the potential causes that generate failures on the same tests of the test suite, assuming there is a single failure-inducing combination. If a combination $C_{i_1,\ldots,i_k}(j_1, \ldots, j_k)$ induces a failure, and appears on rows $m'_1, \ldots, m'_a$ of the test suite, define $\phi(C_{i_1,\ldots,i_k}(j_1, \ldots, j_k))$ as the corresponding set of potential causes that would be generated. That is,

$$\phi(C_{i_1,\ldots,i_k}(j_1, \ldots, j_k)) = \psi_k(\{m'_1, \ldots, m'_a\}).$$

While the number of k-input combinations in $\phi(C_{i_1,\ldots,i_k}(j_1, \ldots, j_k))$ may be large, because of the sparsity of failure-inducing combinations, the question is if there are any single k-input combinations that would produce failures on the exact same rows. If there are, a test engineer cannot distinguish the actual failure-inducing combination from the set of potential causes. Define $\mathrm{Con}(C_{i_1,\ldots,i_k}(j_1, \ldots, j_k))$ as the number of single combinations (distinct from $C_{i_1,\ldots,i_k}(j_1, \ldots, j_k)$) that generate the same potential cause set. So

$$\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k)) = |\phi(C_{i_1,\dots,i_k}(j_1,\dots,j_k)) = \phi(C_{i'_1,\dots,i'_k}(j'_1,\dots,j'_k))|$$

for $C_{i_1,\dots,i_k}(j_1,\dots,j_k) \neq C_{i'_1,\dots,i'_k}(j'_1,\dots,j'_k)$. We refer to $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ as the confounding number for a combination. It is the additional number of combinations that generate the same set of potential causes.

Example 1 Consider a 4-factor binary strength 2 covering array with 5 rows, as shown in Table 2. Let this covering array represent a test suite before the results are collected. Take $C_{1,2}(1,1)$, the level combination of level 1 in both columns 1 and 2. That combination appears only in test 5, and enumerating the two-factor combinations that appear only in test 5, assuming the only failure observed is in test 5, we have

$$\begin{aligned}\phi(C_{1,2}(1,1)) &= \psi_2(\{5\})\\ &= \{C_{1,2}(1,1), C_{1,3}(1,1), C_{1,4}(1,1), C_{2,3}(1,1), C_{2,4}(1,1), C_{3,4}(1,1)\}.\end{aligned}$$

With six items in $\psi_2(\{5\})$, $\mathrm{Con}(C_{1,2}(1,1)) = 5$. Similarly, $C_{1,2}(2,2)$ appears on rows 1 and 2 of the test suite. Assuming $C_{1,2}(2,2)$ is a singular cause and failures are only observed on rows 1 and 2,

$$\begin{aligned}\phi(C_{1,2}(2,2)) = \psi_2(\{1,2\}) = \{&C_{1,2}(2,2), C_{1,4}(2,1), C_{2,4}(2,1), C_{3,4}(2,1),\\ &C_{1,3}(2,1), C_{2,3}(2,1), C_{3,4}(1,2)\}.\end{aligned} \tag{1}$$

However, $C_{1,2}(2,2)$ is the only combination that appears on both rows 1 and 2, so $\mathrm{Con}(C_{1,2}(2,2)) = 0$.

Large values of $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ indicate that it is more difficult to identify $C_{i_1,\dots,i_k}(j_1,\dots,j_k)$ as failure-inducing, as there are more combinations that are equally likely to be failure-inducing under the hierarchy assumption. As there are many $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ values when considering all combinations, as an overall measure one can take the average $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ for all combinations of order k. For a strength t covering array, for combinations with $k < t$ inputs, $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k)) = 0$. This is because for any $C_{i_1,\dots,i_k}(j_1,\dots,j_k)$, if we take any other input, all levels of that input must occur with $C_{i_1,\dots,i_k}(j_1,\dots,j_k)$ by the definition of the strength of a covering array.

For a strength t covering array, $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ is most appropriate for $k = t$. If one considers the $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ for $k > t$, there may be

Table 2 Test suite for Example 1

1	2	3	4	Result
2	2	2	1	•
2	2	1	2	•
2	1	2	2	•
1	2	2	2	•
1	1	1	1	•

combinations of order less than k but greater than or equal to t that result in failures in the same rows. By the hierarchy principle, the lower-order combinations are more likely failure-inducing. Then, if one does want to consider $\mathrm{Con}(C_{i_1,\dots,i_k}(j_1,\dots,j_k))$ with $k > t$, all combinations of order less than k (but $\geq t$) that produce the same set of potential causes should be generated, and one should consider the number of confounding combinations for each order up to k. With this in mind, for $k > t$, a practitioner can evaluate the number of combinations that would need to be examined to localize a particular failure-inducing combination of order k that appears in the test suite.

In Example 1, $|\phi(C_{1,2}(2,2))| = 7$, yet $\mathrm{Con}(C_{1,2}(2,2)) = 0$ as $C_{1,2}(2,2)$ was the only combination appearing on both rows. From a testing perspective, if the only failures occur in tests 1 and 2 with successes elsewhere, a test engineer would start their investigation with the intuitively most likely explanation—namely $C_{1,2}(2,2)$. If it turns out that there is no fault due to such a combination, one would investigate the remaining combinations in the potential cause set. Contrast this with $\phi(C_{1,2}(1,1))$, where a test engineer has six combinations to investigate. Not surprisingly, to reduce the confounding number of a particular combination will require additional test cases, which may not always be feasible. As the number of factors grows, the number of combinations in the set of potential causes can become daunting. Even for a small example as in Example 1, a failure for test case 5 means that there are six two-factor combinations that need to be investigated. The next two sections discuss the use of weights as proposed in Lekivetz and Morgan [18] that can help provide a ranking of potential causes to guide a test engineer in fault localization.

4 Prior Knowledge

If a test suite results in failures, a test engineer is faced with determining a plan of action for reducing the set of potential causes to identify the failure-inducing combination(s). If one follows the hierarchy principle, all potential causes of the same order are treated equally likely to induce a failure, while the heredity principle is not relevant without the notion of main effects. However, test engineers typically know something about the system. In practice, given a set of potential causes, a test engineer often uses their intuition and knowledge of the system, through previous failures and possibly recent changes, to rank the set of potential causes. That is, given a set of potential causes, a test engineer would not consider all the combinations equally likely to induce a failure. This difference between combinations is commonly due to information about the individual inputs or lower-order combinations involved in that combination. In this way, using prior knowledge, we do have a concept akin to the idea of effect heredity—namely particular inputs (possibly individual levels of) are known to be more or less problematic for any combination involving them, even if they do not directly induce a failure on their own. The task of investigating a potential set of causes could be made less burdensome if we could automate the process of ranking potential causes by using prior knowledge that a test engineer may have of the system. The remainder of this section outlines such a method as introduced in Lekivetz and Morgan [18]. The multiplicity of weights is similar to that as described in Bryce et al. [2], which discusses different mechanisms for deriving priorities (weights). Often these weights are arbitrary by nature, based on prioritization from test engineers, but their utility is to achieve a relative ranking of potential causes.

We quantify prior knowledge through weights. Define $w_i(j) \geq 0$ as the weight of level j for input i, for $j = 1, \ldots, s_i$. We use a baseline weight for all $w_i(j)$ to be 1, as in the authors' experience, test engineers find this to be more intuitively appealing when deciding on weights. A $w_i(j)$ should be assigned a value greater than 1 if level j of input i is assumed more likely to be involved in a failure based on prior knowledge. A weight of less than 1 is used for levels that are known to be less problematic. A value of 0 assigned to a $w_i(j)$ is only used in fault characterization, where it is known with absolute certainty that a failure cannot occur due to any interactions that include level j for input i. Such a situation would occur when a test engineer has been able to test that particular input, as would be the case with checking an if statement in a segment of software code. The magnitude of weight $w_i(j)$ can be thought of as how much more (or less) likely level j for input i may lead to a failure compared to a baseline input having weight 1. If all weights are set to 1, all effects of the same order are thought to be equally likely to induce a failure.

We denote the weight for the combination of input i_1 at level j_1 and input i_2 ($i_1 \neq i_2$) at level j_2 by $w_{i_1 i_2}(j_1,j_2)$. The weight of $w_{i_1 i_2}(j_1,j_2)$ is calculated as

$$w_{i_1 i_2}(j_1,j_2) = w_{i_1}(j_1)w_{i_2}(j_2), \tag{2}$$

where $j_1 \in 1, \ldots, s_{i_1}$ and $j_2 \in 1, \ldots, s_{i_2}$. Using baseline weights of 1, values of $w_{i_1 i_2}(j_1,j_2)$ greater than 1 indicate combinations more likely to induce a failure, while a value less than 1 indicates combinations less likely to induce a failure. The specification also implies if both inputs are thought to be more likely involved in inducing a failure, their combination is as well. Equation (2) does not preclude the test engineer from changing the value of $w_{i_1 i_2}(j_1,j_2)$. If a test engineer wishes to be more specific about their knowledge of particular combinations, it can be reflected by using a different value for $w_{i_1 i_2}(j_1,j_2)$ instead of through Eq. (2). This is particularly relevant if one wishes to place a high weight on combinations that have induced failures in the past.

To extend this to three inputs, for distinct inputs i_1, i_2, and i_3 at levels j_1, j_2, and j_3 ($j_a \in 1, \ldots, s_a$), respectively, the weight of the three-input combination, denoted by $w_{i_1 i_2 i_3}(j_1,j_2,j_3)$, can be calculated as

$$w_{i_1 i_2 i_3}(j_1,j_2,j_3) = w_{i_1}(j_1)w_{i_2}(j_2)w_{i_3}(j_3), \tag{3}$$

assuming that Eq. (2) is used for the weights of two-input combinations. As before, prior knowledge may dictate that a different weight should be assigned to $w_{i_1 i_2 i_3}(j_1,j_2,j_3)$ instead of using Eq. (3). If Eq. (2) was not used for any pair of inputs in $\{i_1, i_2, i_3\}$, one can make an adjustment such as using the average or maximum over different configurations of the lower-order weights [18].

Weights for combinations involving more than three inputs can be defined in a similar fashion. In general for $f > 2$ inputs, the weight of a combination is the product of the weights of the individual inputs:

$$w_{i_1,i_2,\ldots,i_f}(j_1,j_2,\cdots,j_f) = w_{i_1}(j_1)w_{i_2}(j_2)\cdots w_{i_f}(j_f),$$

with adjustments as deemed necessary by the test engineer. We now discuss how these weights can be used in evaluating a covering array.

While the SUT is deterministic, before running a test suite, it is unknown which, if any, combinations are failure-inducing. Let $X_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ take a value of 1 if $C_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ induces a fault, and 0 otherwise. Before running any tests, we treat $X_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ as a Bernoulli random variable. For convenience, we use $P(C_{i_1,\ldots,i_k}(j_1,\cdots,j_k))$ to denote $P(X_{i_1,\ldots,i_k}(j_1,\cdots,j_k)=1)$. This reflects the prior belief that $C_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ will be failure-inducing, whereby any test involving $C_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ will result in a failure. Define

$$P(X_{i_1,\ldots,i_k}(j_1,\cdots,j_k)=1) = w_{i_1,\ldots,i_k}(j_1,\cdots,j_k)/a$$

where a is sufficiently large. While $a \geq \max(w_{i_1,\ldots,i_k}(j_1,\cdots,j_k))$ for all possible combinations of size k, in cases where the sparsity assumption holds, a reasonable value of a is $\sum w_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ over all combinations of size k. That is, if we take the expected value over all combinations of size k, the expected number of failure-inducing combinations is 1. We assume throughout that $X_{i_1,\ldots,i_k}(j_1,\cdots,j_k)'s$ are independent, and if a particular subset of fewer than k inputs is failure-inducing, the fault localization problem needs to address the smaller subset first. Note that $P(X_{i_1,\ldots,i_k}(j_1,\cdots,j_k))=0)$ is close to 1, as it is expected that almost every combination is not failure-inducing.

5 Evaluation of Covering Arrays Using Weights

If combinations having higher weight are more likely to induce failures, then a test engineer would like to have those combinations appear in the test suite. For a strength t covering array, Lekivetz and Morgan [17] propose the weighted k-coverage ($k > t$) as a criterion to be used in construction. The weighted k-coverage of a covering array is defined as

$$\text{coverage}_w(k) = \frac{\sum I(w_{i_1,\ldots,i_k}(j_1,\cdots,j_k))}{\sum w_{i_1,\ldots,i_k}(j_1,\cdots,j_k)}$$

where the summation is over all possible k-column projections and associated levels and $I(w_{i_1,\ldots,i_k}(j_1,\cdots,j_k))$ takes a value of $w_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ if the combination $C_{i_1,\ldots,i_k}(j_1,\cdots,j_k)$ appears in the test suite and 0 otherwise. Maximizing this measure during construction or as a post-construction optimization step increases the use of combinations more likely to induce a failure. Since preference is given to level combinations having higher weight, k-factor combinations involving them are likely to appear more than once, aiding in the fault localization problem. One method of post-construction optimization is to use *don't care cells*, for which any eligible level for that column that contains the cell can be used for that cell without altering the t-coverage of the covering array [4]. Kim, Jang, and Anderson-Cook [12] investigate

Table 3 Strength 2 binary covering array for Example 2

1	2	3	4
2	2	2	1
2	2	1	2
2	1	2	2
1	2	2	2
1	1	1	1
2	1	1	2

Table 4 Strength 2 binary covering array with don't care cells marked in Example 2

1	2	3	4
2	2	2	1
2	2	1	2
2	1	2	2
1	2	2	2
1	1	1	1
*	*	*	*

some criteria for evaluating covering arrays and the use of don't care cells (wild card entries). The appeal of don't care cells is that a test engineer can set those particular entries in the test suite to any valid value they choose. The choice of values is typically used to optimize $t + i$ coverage ($i \geq 1$) of the covering array. However, the don't care cells can be used to maximize the weighted $t + 1$-coverage [17]. When all the weights are 1, using don't care cells to maximize the weighted $t + 1$-coverage is equivalent to maximizing the $t + 1$ coverage. We demonstrate through a simple example.

Example 2 Table 3 provides a strength 2 covering array with 4 binary input factors in 6 runs. Table 4 presents Table 3 with the last row identified as having all positions as don't care cells (denoted by *) [10]. That is, the cells of the last row can take on any value and still remain a strength 2 covering array. Consider the weighting scheme $w_1(1) = 4$ and $w_2(1) = 4$, with all remaining weights set to 1. That is, the test engineer believes that level 1 of inputs 1 and 2 are more likely to be involved in combinations that induce failures. Using the weighted 3-coverage criterion, considering all possible combinations of values for the don't care cells, the values (1, 1, 2, 2) would maximize weighted 3-coverage. Using this weighting scheme, without taking into account the don't care row, the baseline weighted 3-coverage is 0.529. When the row of don't care cells is set to (2, 2, 1, 1), weighted 3-coverage increases marginally to 0.543. However, if the row of don't care cells is set to (1, 1, 1, 2), weighted 3-coverage increases to 0.7 and when set to (1, 1, 2, 2), weighted 3-coverage attains its optimal value of 0.757. Similarly, using $w_3(1) = 4$ and $w_4(1) = 4$ with the remaining weights set to 1, the row of don't care cells should be set to (2, 2, 1, 1) to ensure optimal weighted 3-coverage.

One of the advantages of using the weighted coverage criterion is that when able to make changes through don't care cells or added rows, the confounding numbers of higher weighted, more likely failure-inducing, combinations are more likely to decrease. In Example 2, prior to changing the don't care cells, $\text{Con}(C_{12}(1,1)) = 4$, whereas setting the don't care cells to (1, 1, 2, 2), $\text{Con}(C_{12}(1,1)) = 0$. As it turns out, lowering these confounding numbers tends to be done by increasing the number of occurrences of higher-weighed combinations, which in turn aids in isolating those combinations if they do induce a failure. In the next section, we introduce how weights can aid in isolating failure-inducing combinations when failures occur in a test suite.

6 Fault Localization

In this section, we discuss how one can examine a combinatorial test suite when failures are observed and use prior information for guidance in localizing failure-inducing combinations. That is, given failures in rows $m_1, \ldots, m_b$, a test engineer needs guidance to isolate potential causes in $\psi_k(\{m_1, \ldots, m_b\})$. The value of k is the smallest such value where the set of potential causes is non-empty, either by successes in the original test suite, or follow-up testing based on the original test suite. Not surprisingly, the frequency and location of the potential causes play a role in fault localization.

The purpose here is to provide a ranking of the potential causes to aid the test engineer in which combinations to investigate first. Assume that we observe failures in rows $m_1, \ldots, m_b$. In our treatment of the problem, we focus only on those combinations that induce a failure—namely $X_{i_1,\ldots,i_k}(j_1, \ldots, j_k) = 1$. While one could consider multiple configurations (i.e., combinations of $X_{i_1,\ldots,i_k}(j_1, \ldots, j_k)'s$ being 0 and 1), of the potential causes maintaining failures only being observed on rows $m_1, \ldots, m_b$, the fact that any given $P(X_{i_1,\ldots,i_k}(j_1, \ldots, j_k) = 0)$ is close to 1, makes it impractical to consider these cases beyond the simplest explanations. For example, if we observe a failure on one row, we assume there is only one failure-inducing combination rather than all possibilities with one or more failure-inducing combinations. In addition, if a failure-inducing combination is found and fixed, so long as a test engineer reruns the particular test with all inputs again, the remaining failure-inducing combination(s) will still be present and the test will still fail.

We first consider the simplest case, where there is a single failure in the test suite on row m_1. Given the set of potential causes, $\psi_k(\{m_1\})$, we can use the weights as described in Sect. 4 to rank each of the combinations. Using the weights, the probability that a particular combination is failure-inducing is

$$P[C_{i_1,\ldots,i_k}(j_1, \ldots, j_k) | \psi_k(\{m_1\})] = \frac{w_{i_1,\ldots,i_k}(j_1, \ldots, j_k)}{\sum\limits_{C_{a_1,\ldots,a_k}(b_1,\ldots,b_k) \in \psi_k(\{m_1\})} w_{a_1,\ldots,a_k}(b_1, \ldots, b_k)}, \tag{4}$$

where $P[C_{i_1,\ldots,i_k}(j_1, \ldots, j_k) | \psi_k(\{m\})] = 0$ if $C_{i_1,\ldots,i_k}(j_1, \ldots, j_k) \notin \psi_k(\{m\})$. Simply put, given a set of potential causes, if we focus on a single failure, the likelihood

Table 5 Weights assigned to each input in Example 3

Input	Levels	Weights
Diagnostic Plot	Checked, unchecked	1, 2
Density Curve	Checked, unchecked	1, 1
Goodness of Fit	Checked, unchecked	1, 1
Save Fit Quantiles	Checked, unchecked	1, 2
Save Dens Formula	Checked, unchecked	3, 1
Save Spec Limit	Checked, unchecked	1, 1

Table 6 Test suite for Example 3

Result	Diagnostic Plot	Density Curve	Goodness of Fit	Save Fit Quantiles	Save Dens Formula	Save Spec Limit
Pass	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked
Pass	Checked	Checked	Checked	Checked	Checked	Checked
Fail	Checked	Unchecked	Checked	Unchecked	Checked	Unchecked
Pass	Checked	Checked	Unchecked	Checked	Unchecked	Unchecked
Pass	Unchecked	Unchecked	Unchecked	Checked	Checked	Checked
Pass	Unchecked	Checked	Checked	Unchecked	Unchecked	Checked

of a particular combination inducing that failure is the weight for that combination divided by the sum of the weights for all combinations in $\psi_k(\{m\})$. There are two underlying assumptions in using Eq. (4): (1) that the failure is not caused by a combination of greater than k inputs, and (2) if it is due to a combination of k inputs, there is only one cause for the failure in test case m. In practice, this follows the approach test engineers intuitively follow in fault localization. Failure-inducing combinations empirically adhere to the hierarchy principle, and a test engineer will typically eliminate all combinations of k inputs as potential causes before considering $k + 1$ inputs. Alternatively, on the occasions that the most likely potential causes for k inputs have been eliminated, a test engineer may use the same approach for $k + 1$ inputs. In regard to (2) while there is a chance that such a situation can occur, such occurrences are rare in a well-tested SUT.

Example 3 Test engineers for a commercially available software product want to test the preferences system of a particular routine in the software. There are six different preferences that can be either checked or unchecked (on/off) as shown in Fig. 1. Table 5 provides weights for each input and associated levels that test engineers agreed on given prior knowledge of the system. A failure in a test case could be incorrect calculations of statistics, graphical elements not displaying correctly, to the software crashing and shutting down prematurely. Examining Table 5, the test engineers have set the weight of checked level of "Save Dens Formula" to be 3, indicating it is known to be problematic, as are the unchecked level of "Diagnostic Plot" and unchecked level of "Save Fit Quantiles," but to a lesser extent with a weight of 2. The remaining weights are set at the baseline value of 1. Table 6 provides a test suite that is a strength 2 covering array, for the inputs in Table 5. The outcome of each test case is presented in the first column. Using the outcome of the test suite, Table 7 presents the potential causes, along with the $w_{i_1 i_2}(j_1, j_2)$ values and their relative probability. Instead of treating all combinations equally likely to have caused the failure, the most likely candidate is $C_{45}(2, 1)$, followed by $C_{56}(1, 2)$ and $C_{14}(1, 2)$. The remaining potential causes are equally likely. This allows the test engineers to first focus on "Save Fit Quantiles" at unchecked and "Save Dens Formula" at checked, and if $C_{45}(2, 1)$ is not failure-inducing, then $C_{56}(1, 2)$ and $C_{14}(1, 2)$ would be the next candidates for investigation.

Example 3 is simple, both in terms of the number of tests and the number of inputs, yet there were still six potential causes for the single failure. Using weights is a way of ranking the potential causes in an automated way that accounts for a test engineer's experience with the system. In Example 3, based on the weights the test engineers used, any combination involving "Save Dens Formula" as checked would be the natural place to look for the cause of the failure given the heredity principle. As the number of inputs grows and test suites become larger, the resulting increase in the number of potential causes is an issue that test engineers have to deal with. The ranking provided by this method makes such situations manageable.

When there is more than one failure, fault localization requires additional care. This is due to the possibility that there may be more than one failure-inducing combination. There may be single combinations that could cause all the failures, which would be the most likely causes. But one may still have a set of multiple combinations that would result in failures of the same rows, albeit less likely. A test engineer may be interested in the likelihood that these combinations might be the cause, particularly when using prior information. Lekivetz and Morgan [18] considered the case of two failures.

Assume that failures have been observed on tests $m_1, \ldots, m_b$ of the test suite. As in Sect. 3, let $\psi_k(\{m_1, \ldots, m_b\})$ be the set of potential causes for k inputs if a failure was to occur in rows $m_1, \ldots, m_b$ of the test suite with all other tests passing. The test engineer wants to ascertain which combination(s) are most likely to have induced the observed failures. Given the set of potential causes $\psi_k(\{m_1, \ldots, m_b\})$, we need to consider sets of potential cause(s) from $\psi_k(\{m_1, \ldots, m_b\})$ that minimally account for failures on all of the rows $m_1, \ldots, m_b$ without any redundancies. Note that not all sets from

Table 7 Probabilities and weights assigned to potential causes $C_{i_1 i_2}(j_1, j_2)$ in Example 3

Input (i)	Failure level (j)	Weight	Probability
Diagnostic Plot (1), Density Curve (2)	Checked (1), unchecked (2)	1	1/14
Diagnostic Plot (1), Save Fit Quantiles (4)	Checked (1), unchecked (2)	2	1/7
Density Curve (2), Goodness of Fit (3)	Unchecked (2), checked (1)	1	1/14
Goodness of Fit (3), Save Spec Limit (6)	Checked (1), unchecked (2)	1	1/14
Save Fit Quantiles (4), Save Dens Formula (5)	Unchecked (2), checked (1)	6	3/7
Save Dens Formula (5), Save Spec Limit (6)	Checked (1), unchecked (2)	3	3/14

$\psi_k(\{m_1,\ldots,m_b\})$ are valid. For a given set of combinations from $\psi_k(\{m_1,\ldots,m_b\})$, if none is observed in row m_i, then the failure rows (and set of potential causes) cannot be generated by that subset. Using $X_{i_1,\ldots,i_k}(j_1,\ldots,j_k)$ as described in Sect. 4, we want

$$P(X_{i_1,\ldots,i_k}(j_1,\ldots,j_k)=1,\ldots,=X_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)=1|\psi_k(\{m_1,\ldots,m_b\})), \tag{5}$$

where $C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)$ being failure-inducing would result in observing failures only in rows $m_1,\ldots,m_b$.

Once the set of subsets for the $X_{i_1,\ldots,i_k}(j_1,\ldots,j_k)=1$ has been determined, one must determine the appropriate probabilities. As some combinations may occur on more than one row, instead of writing the probability for each row (i.e., where the failures occur), we instead present it in terms of the potential causes that cover all failures in rows $m_1,\ldots,m_b$. If there is a single combination in that set, Eq. (5) then becomes

$$P(C_{i_1,\ldots,i_k}(j_1,\ldots,j_k)|\psi_k(\{m_1,\ldots,m_b\})),$$

while for more than one potential cause,

$$P(C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)|\psi_k(\{m_1,\ldots,m_b\})).$$

That is, what is the probability that $C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)$ are all failure-inducing combinations, given that failures have been observed in rows $m_1,\ldots,m_b$. For potential failure-inducing combinations that occur on multiple rows, intuitively one expects the associated probability to be higher than that of distinct failure-inducing combinations occurring on different rows. To quantify this, we need to establish a likelihood of any particular combination inducing a failure, which we denote by $P[C_{i_1,\ldots,i_k}(j_1,\ldots,j_k)]$. We use the weighting scheme described in Sect. 4 (i.e., one expected failure-inducing combination),

$$P[C_{i_1,\ldots,i_k}(j_1,\ldots,j_k)]=\frac{w_{i_1\cdots i_k}(j_1,\ldots,j_k)}{\sum w_{a_1\cdots a_k}(b_1,\ldots,b_k)}. \tag{6}$$

Given failures have been observed on certain rows, the likelihood of particular combinations inducing a failure is either 0, since they passed elsewhere, or much higher since the set of potential causes is much smaller than the set of all possible combinations.

Let θ be the set of combinations that can generate in $\psi_k(\{m_1,\ldots,m_b\})$ that would minimally produce failures in $m_1,\ldots,m_b$. As before, we assume failure-inducing combinations are independent. Given a $C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)$ in θ,

$$\begin{aligned} &P(C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)|\psi_k(\{m_1,\ldots,m_b\}))\\ &\quad=\frac{P(C_{i_1,\ldots,i_k}(j_1,\ldots,j_k),\ldots,C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k)\psi_k(\{m_1,\ldots,m_b\}))}{P(\psi_k(\{m_1,\ldots,m_b\}))}\\ &\quad=\frac{P(C_{i_1,\ldots,i_k}(j_1,\ldots,j_k))\cdots P(C_{i'_1,\ldots,i'_k}(j'_1,\ldots,j'_k))}{P(\psi_k(\{m_1,\ldots,m_b\}))} \end{aligned} \tag{7}$$

Table 8 Weights assigned to each input in Example 4

Input	Levels	Weights
Diagnostic Plot	Checked, unchecked	2, 2
Density Curve	Checked, unchecked	2, 2
Goodness of Fit	Checked, unchecked	1, 1
Save Fit Quantiles	Checked, unchecked	1, 1
Save Dens Formula	Checked, unchecked	3, 2
Save Spec Limit	Checked, unchecked	1, 1

Table 9 Test suite for Example 4

Result	Diagnostic Plot	Density Curve	Goodness of Fit	Save Fit Quantiles	Save Dens Formula	Save Spec Limit
Pass	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked
Pass	Checked	Checked	Checked	Checked	Checked	Checked
Pass	Checked	Unchecked	Checked	Unchecked	Checked	Unchecked
Fail	Checked	Checked	Unchecked	Checked	Unchecked	Unchecked
Fail	Unchecked	Unchecked	Unchecked	Checked	Checked	Checked
Pass	Unchecked	Checked	Checked	Unchecked	Unchecked	Checked

where the numerator of Eq. (7) can be derived by noting that the sets of combinations in θ are chosen because they can generate $\psi_k(\{m_1, \dots, m_b\})$.

To calculate $P(\psi_k(\{m_1, \dots, m_b\})$, we take the summation over the set of subsets in θ,

$$P(\psi_k(\{m_1, \dots, m_b\}) = \sum_{\theta} P(C_{i_1,\dots,i_k}(j_1, \dots, j_k)) \cdots P(C_{i'_1,\dots,i'_k}(j'_1, \dots, j'_k)), \tag{8}$$

where the elements in θ may have only one combination.

Example 4 We return to the set of inputs in Example 3. In this case, the test engineers have decided on the set of weights given in Table 8. The same strength 2 covering array is used to test the system, but the results are different as shown in Table 9. No single input is failure-inducing, so two-input combinations must be considered. The summation of all $w_{i_1,i_2}(j_1,j_2)$ is 146. Using Eq. (6), the potential causes for each row and the associated probabilities are given in Table 10. Note that the combination “Goodness of Fit” at unchecked and “Save Fit Quantiles” at checked ($C_{34}(2, 1)$) appears in both test cases as a potential cause. This implies that in θ, $C_{34}(2, 1)$ will be a combination on its own, while the other elements of θ will contain one of the potential causes from test case 4 only, and one from test case 5 only. Using Eq. (8), we can partition it into $P(C_{34}(2, 1)))$, plus the sum of the product of the remaining pairs of potential causes.

Table 10 Potential causes for each failure in Example 4

Test case 4			Test case 5		
$C_{i_1i_2}(j_1,j_2)$	$w_{i_1i_2}(j_1,j_2)$	$P[C_{i_1i_2}(j_1,j_2)]$	$C_{i_1i_2}(j_1,j_2)$	$w_{i_1i_2}(j_1,j_2)$	$P[C_{i_1i_2}(j_1,j_2)]$
$C_{34}(2,1)$	1	1/146	$C_{34}(2,1)$	1	1/146
$C_{13}(1,2)$	2	2/146	$C_{14}(2,1)$	2	2/146
$C_{15}(1,2)$	4	4/146	$C_{15}(2,1)$	6	6/146
$C_{23}(1,2)$	2	2/146	$C_{24}(2,1)$	2	2/146
$C_{26}(1,2)$	2	2/146	$C_{26}(2,1)$	2	2/146
$C_{45}(1,2)$	2	2/146	$C_{35}(2,1)$	3	3/146
$C_{46}(1,2)$	1	1/146	$C_{36}(2,1)$	1	1/146

$$P(\psi_2(\{4,5\})) = \frac{1}{146} + \frac{(13)(16)}{146^2} = \frac{354}{146^2} \tag{9}$$

Table 11 presents the joint probability for potential causes considering both test cases. With only two failures, one can examine the marginal probabilities. They are shown in Table 11 in the last column for test case 4 and the last row for test case 5.

As demonstrated in Example 4, even when the initial weight of a particular combination is small, when there is a potential cause (or causes) that appears for multiple failures, it tends to be the most likely potential cause compared to independent causes for different failures. This is intuitively what one would expect if failures are rare. From a practical standpoint, it is often preferable to present potential causes from multiple test cases separately from the unique occurrences. That is, do a weighted comparison of the smallest sets in θ first. If it is discovered that none of the multiple-test case potential causes are failure-inducing, the test engineer can perform a weighted analysis on the remaining potential causes unique to each test case. If there are more than two failures, this approach is typically more tractable than generating the joint probability mass function over all sets in θ. We finish this section with an example where three failures are observed.

Table 11 Probabilities for potential causes for each failure test case in Example 4

	$C_{34}(2,1)$	$C_{14}(2,1)$	$C_{15}(2,1)$	$C_{24}(2,1)$	$C_{26}(2,1)$	$C_{35}(2,1)$	$C_{36}(2,1)$	
$C_{34}(2,1)$	$\frac{146}{354}$	0	0	0	0	0	0	146/354
$C_{13}(1,2)$	0	$\frac{4}{354}$	$\frac{12}{354}$	$\frac{4}{354}$	$\frac{4}{354}$	$\frac{6}{354}$	$\frac{2}{354}$	32/354
$C_{15}(1,2)$	0	$\frac{8}{354}$	$\frac{24}{354}$	$\frac{8}{354}$	$\frac{8}{354}$	$\frac{12}{354}$	$\frac{4}{354}$	64/354
$C_{23}(1,2)$	0	$\frac{4}{354}$	$\frac{12}{354}$	$\frac{4}{354}$	$\frac{4}{354}$	$\frac{6}{354}$	$\frac{2}{354}$	32/354
$C_{26}(1,2)$	0	$\frac{4}{354}$	$\frac{12}{354}$	$\frac{4}{354}$	$\frac{4}{354}$	$\frac{6}{354}$	$\frac{2}{354}$	32/354
$C_{45}(1,2)$	0	$\frac{4}{354}$	$\frac{12}{354}$	$\frac{4}{354}$	$\frac{4}{354}$	$\frac{6}{354}$	$\frac{2}{354}$	32/354
$C_{46}(1,2)$	0	$\frac{2}{354}$	$\frac{6}{354}$	$\frac{2}{354}$	$\frac{2}{354}$	$\frac{3}{354}$	$\frac{1}{354}$	16/354
	146/354	26/354	78/354	26/354	26/354	39/354	13/354	p

Table 12 Test suite for Example 5

Result	Diagnostic Plot	Density Curve	Goodness of Fit	Save Fit Quantiles	Save Dens Formula	Save Spec Limit
Pass	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked	Unchecked
Fail	Checked	Checked	Checked	Checked	Checked	Checked
Pass	Checked	Unchecked	Checked	Unchecked	Checked	Unchecked
Fail	Checked	Checked	Unchecked	Checked	Unchecked	Unchecked
Pass	Unchecked	Unchecked	Unchecked	Checked	Checked	Checked
Pass	Unchecked	Checked	Checked	Unchecked	Unchecked	Checked
Fail	Checked	Checked	Unchecked	Unchecked	Unchecked	Checked

Table 13 Probabilities for potential causes in θ in Example 5

Potential cause(s)	Probability	Potential cause(s)	Probability
$C_{12}(1,1)$	292/353	$C_{15}(1,2), C_{25}(1,1)$	12/353
$C_{13}(1,2), C_{14}(1,1)$	2/353	$C_{15}(1,2), C_{34}(1,1)$	2/353
$C_{13}(1,2), C_{16}(1,1)$	2/353	$C_{16}(1,1), C_{23}(1,2)$	2/353
$C_{13}(1,2), C_{24}(1,1)$	2/353	$C_{16}(1,1), C_{24}(1,1)$	2/353
$C_{13}(1,2), C_{25}(1,1)$	6/353	$C_{16}(1,1), C_{26}(1,2)$	2/353
$C_{13}(1,2), C_{34}(1,1)$	1/353	$C_{16}(1,1), C_{45}(1,2)$	2/353
$C_{14}(1,1), C_{15}(1,2)$	4/353	$C_{16}(1,1), C_{46}(1,2)$	1/353
$C_{14}(1,1), C_{16}(1,1)$	2/353	$C_{23}(1,2), C_{24}(1,1)$	2/353
$C_{14}(1,1), C_{23}(1,2)$	2/353	$C_{23}(1,2), C_{25}(1,1)$	6/353
$C_{15}(1,2), C_{16}(1,1)$	4/353	$C_{23}(1,2), C_{34}(1,1)$	1/353
$C_{15}(1,2), C_{24}(1,1)$	4/353		

Example 5 We return to the set of inputs in Example 3, using the weights from Example 4 given in Table 8. In this example, an additional test has been added to the test suite, given in Table 12. Failures are observed in tests 2, 4, and 7, with the remaining tests passing. Table 13 provides the set(s) of potential causes in θ that would generate $\psi_2(\{2,4,7\})$ and associated probabilities. In this example, the combination of diagnostic plot and density curve both checked would be a test engineer's first consideration.

When there are three or more failures in a test suite, the number of sets of combinations in θ can be very large. As seen in the examples, the elements of θ that contain more input combinations are less likely to be the cause of the failures, as would be expected given the sparsity principle. If θ has too many elements, a practitioner could first consider those sets in θ having the smallest number of elements.

7 Discussion

In this article, we have discussed the connection between the seemingly disparate concepts of failure-inducing combinations in a complex engineered system and the principles of factorial effects that are fundamental to constructing designs for factorial effects. We have demonstrated how prior information about a system can be quantified through the use of weights, aiding in both the evaluation and analysis of covering arrays. In the construction of covering arrays, the use of weights will result in a covering array that will have a higher likelihood of uncovering faults in the system, while for analysis, they aid in fault localization. An area of future research is the use of weights to augment a covering array when a test engineer has the budget for additional test cases. These extra test cases can reduce the number of likely potential causes when failures occur and can therefore provide a happy medium between optimal size covering arrays and error-locating arrays [5]. Alternatively, our method of computing weights could be used by algorithms that construct biased covering arrays [2] or to encode prior information for a test suite prioritization technique [8].

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Box G, Tyssedal J (1996) Projective properties of certain orthogonal arrays. Biometrika 83(4):950–955
2. Bryce RC, Colbourn CJ (2006) Prioritized interaction testing for pair-wise coverage with seeding and constraints. Inf Softw Technol 48(10):960–970. https://doi.org/10.1016/j.infsof.2006.03.004 **(Advances in model-based testing)**
3. Chen T, He T, Benesty M, Khotilovich V, Tang Y (2015) Xgboost: extreme gradient boosting. R package version 0.4-2, pp 1–4
4. Colbourn CJ, Martirosyan SS, Mullen GL, Shasha D, Sherwood GB, Yucas JL (2006) Products of mixed covering arrays of strength two. J Comb Des 14(2):124–138
5. Colbourn CJ, McClary DW (2008) Locating and detecting arrays for interaction faults. J Comb Optim 15(1):17–48. https://doi.org/10.1007/s10878-007-9082-4
6. Colbourn CJ, Syrotiuk VR (2016) Coverage, location, detection, and measurement. In: IEEE ninth international conference on software testing, verification and validation workshops (ICSTW), pp 19–25. https://doi.org/10.1109/ICSTW.2016.38
7. Dalal SR, Mallows CL (1998) Factor-covering designs for testing software. Technometrics 40(3):234–243. https://doi.org/10.1080/00401706.1998.10485524
8. Elbaum S, Rothermel G, Kanduri S, Malishevsky AG (2004) Selecting a cost-effective test case prioritization technique. Softw Qual J 12(3):185–210. https://doi.org/10.1023/B:SQJO.0000034708.84524.22
9. Ghandehari LS, Lei Y, Kung D, Kacker R, Kuhn R (2013) Fault localization based on failure-inducing combinations. In: IEEE 24th international symposium on software reliability engineering (ISSRE), pp 168–177. https://doi.org/10.1109/ISSRE.2013.6698916
10. Gonzalez-Hernandez L, Torres-Jiménez J, Rangel-Valdez N (2011) An exact approach to maximize the number of wild cards in a covering array. In: Batyrshin I, Sidorov G (eds) Advances in

artificial intelligence. MICAI 2011. Lecture notes in computer science, vol 7094. Springer, Berlin, Heidelberg

11. Katona GO (1973) Two applications (for search theory and truth functions) of sperner type theorems. Period Math Hung 3(1–2):19–26
12. Kim Y, Jang DH, Anderson-Cook CM (2017) Selecting the best wild card entries in a covering array. Qual Reliab Eng Int 33(7):1615–1627
13. Kleitman DJ, Spencer J (1973) Families of k-independent sets. Discret Math 6(3):255–262
14. Kuhn DR, Okum V (2006) Pseudo-exhaustive testing for software. In: 30th Annual IEEE/NASA software engineering workshop. IEEE, pp 153–158
15. Kuhn DR, Reilly MJ (2002) An investigation of the applicability of design of experiments to software testing. In: Proceedings 27th annual NASA Goddard/IEEE software engineering workshop. IEEE, pp 91–95
16. Kuhn DR, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 30(6):418–421. https://doi.org/10.1109/TSE.2004.24
17. Lekivetz R, Morgan J (2018) Evaluation and construction of covering arrays utilizing prior information. In: IEEE international symposium on software reliability engineering workshops (ISSREW). IEEE, pp 132–133
18. Lekivetz R, Morgan J (2018) Fault localization: analyzing covering arrays given prior information. In: IEEE international conference on software quality, reliability and security companion (QRS-C). IEEE, pp 116–121
19. Morgan J (2018) Combinatorial testing: an approach to systems and software testing based on covering arrays. In: Kenett RS, Ruggeri F, Faltin FW (eds) Analytic methods in systems and software testing. Wiley, pp 131–158
20. Qu X, Cohen MB, Rothermel G (2008) Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the 2008 international symposium on software testing and analysis. ACM, pp 75–86
21. Rényi A (2007) Foundations of probability. Courier Corporation, North Chelmsford
22. Wu CJ, Hamada MS (2011) Experiments: planning, analysis, and optimization, vol 552. Wiley, Hoboken

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.