Statistical Theory and Related Fields

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tstf20

Combinatorial testing: using blocking to assign test cases for validating complex software systems

Ryan Lekivetz & Joseph Morgan

To cite this article: Ryan Lekivetz & Joseph Morgan (2021) Combinatorial testing: using blocking to assign test cases for validating complex software systems, Statistical Theory and Related Fields, 5:2, 114-121, DOI: 10.1080/24754269.2021.1904095

To link to this article: https://doi.org/10.1080/24754269.2021.1904095

Published online: 06 Apr 2021.

STATISTICAL THEORY AND RELATED FIELDS
2021, VOL. 5, NO. 2, 114–121
https://doi.org/10.1080/24754269.2021.1904095

Taylor & Francis
Taylor & Francis Group

Combinatorial testing: using blocking to assign test cases for validating complex software systems

Ryan Lekivetz and Joseph Morgan

JMP Division of SAS, Cary, NC, USA

ABSTRACT
Testing complex software systems is an extraordinarily difficult task. Test engineers are faced with the challenging prospect of ensuring that a software system satisfies its requirements while working within a strict budget. Choosing a test suite for such an endeavour can be framed as a design of experiments problem. Combinatorial testing is a software testing methodology that may be viewed as a design of experiments approach to addressing the software testing challenge. We extend this methodology by introducing the concept of blocking factors for a test suite. We provide an example, using an open source software library, to illustrate our extension. Advantages of considering blocks are discussed, both in the design as well as after test execution, when fault localisation may be necessary.

ARTICLE HISTORY
Received 14 September 2020
Revised 12 March 2021
Accepted 12 March 2021

KEYWORDS
Combinatorial testing; covering arrays; design of experiments; software engineering

1. Introduction

With the rapid adoption of machine learning algorithms across an increasingly broad set of disciplines, the following question arises with increasing frequency: 'Is this algorithm doing what I expect?' Ensuring that an algorithm, or a software system, meets its requirements is referred to by software engineers as the 'software validation problem' (Myers et al., 2004). Consider XGBoost, a widely used open source gradient boosting library (Chen & Guestrin, 2016), like many other complex software systems, the algorithms in the XGBoost library are developed independently by several different development groups and then integrated into the library. Although we may assume that the individual algorithms have been rigorously tested by individual development groups, it may be unreasonable to expect that validation of the integrated library is as rigorous. Furthermore, the XGBoost library provides a set of hyperparameters that allows users to configure a particular algorithm in the library at run time. It may also be unreasonable to expect that the configuration space of the XGBoost library is as rigorously validated as individual algorithms. At the time of writing, the current version of XGBoost offers 34 hyperparameters, some of which are continuous values and so the configuration space is infinitely large. Consequently, validating the possible configurations of the XGBoost library is an extraordinarily challenging problem. This process of validating the components, or configurations, of a software system is referred to by software engineers as the 'configuration testing problem' and is an area of active research interest in the software engineering community (Cohen et al., 2007). As it turns out, approaching this problem as a designed experiment can be an effective way of addressing the problem. We can think of the response for these problems as a binary response indicating whether a failure was observed for each test case. Typically, this response is deterministic, so a test case that induces a failure will consistently do so.

To further develop this idea, let us consider the case where for some software system, several test engineers are available to validate the system. A natural question that arises is how to effectively distribute the testing effort among the test engineers. One strategy would be to partition the components of the system into groups, assign each test engineer a group, and have each test engineer test the set of components assigned to them, placing emphasis on individual components one at a time. Whereas this strategy may ensure that individual components work as intended, in effect, it treats each test engineer as an independent agent and so fails to take into account efficiencies that could accrue if the entire testing effort was treated as a designed experiment. As a result, although such a testing strategy may appear reasonable, it is not an efficient strategy.

To gain some insight into how to proceed, let us start with an adage from the software engineering community, attributed to Boris Beizer, who succinctly stated, 'Bugs lurk in corners and congregate at boundaries.' (Beizer, 2003) The boundaries that Beizer refers to reflect software failures induced by edge cases, whereas the corners are the values of two or more inputs that induce a failure. With this practical adage in mind,

CONTACT Ryan Lekivetz ryan.lekivetz@jmp.com JMP Division of SAS, 100 SAS Institute Drive, Cary, NC, USA

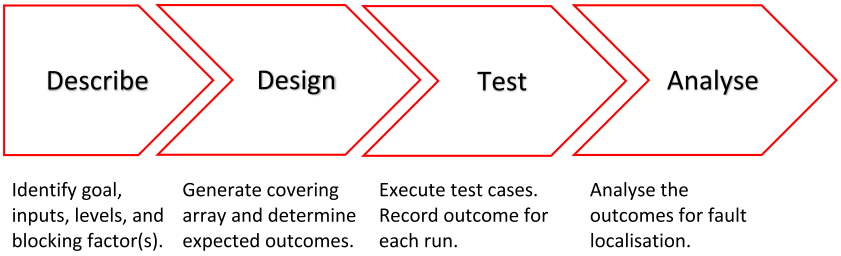

Figure 1. Framework for testing a complex system.

a test engineer should seek to identify test cases that involve edge cases as well as those that cover as many corners as possible. If one approaches the concept of failure-inducing combinations as factorial effects in a designed experiment, the hierarchical ordering principle and effect sparsity have their counterparts. As discussed in Lekivetz and Morgan (2020), combinations involving fewer inputs are more likely to induce a failure, and the assumption of the existence of only a few failure-inducing combinations within the system is reasonable if sufficient testing and bug-fixing has been previously done. The counterpart to the hierarchical ordering principle has been empirically shown for a variety of complex software systems (Kuhn et al., 2004). It turns out that test engineers can construct a set of test cases to cover combinations of inputs, if they make use of a special type of design known as a covering array (Dalal & Mallows, 1998). For such designs, the strength t of the covering array ensures that every combination involving t or fewer inputs will be covered. This approach, where covering arrays are used as the underlying construct for testing, is referred to as combinatorial testing (Morgan, 2018).

In this paper we propose blocking factors as an extension to the combinatorial testing approach and show how this idea can enhance the effectiveness of the approach. While blocking is a core principle in the design of experiments, this is not the case in combinatorial testing. We begin by outlining a combinatorial testing methodology that is conceptually similar to how one would approach a designed experiment. In this methodology, blocking factors are defined in the same step as the inputs. In Section 3, we introduce notation for covering arrays and describe how to accommodate blocking factors. Sections 4 and 5 illustrate how our methodology may be used to validate the hyperparameters of the XGBoost software library and the usefulness of the blocking approach when a failure is induced. The paper concludes with a discussion.

2. Methodology

The process of validating complex software systems involves several activities that can usually be readily organised into a series of distinct steps or phases. It turns out that these phases are conceptually similar to the phases that one would follow in designing an experiment. Given this observation, we propose the workflow summarised in Figure 1 as a framework for validating complex software systems.

Describe: In the describe phase, the intent is to identify the overall goal of the testing effort as well as the inputs, and associated levels, of the software system being tested. For test engineers, the goal of testing is generally to provide sufficient evidence to allow them to make assertions about the fitness for use of a software system. For the describe phase, we propose a more precise formulation of this general goal that is based on the idea of failure-inducing combinations. That is, test engineers should determine the t-way combinations that they want covered by the suite of test cases. Such a goal allows test engineers to assess whether the software system being tested is free of faults due to t-way combinations. Of course, a test engineer wants t as large as possible, but testing budget ultimately determines t. In fact, in many situations t is chosen to be two. Next, inputs should be identified and then, for each input, the set of allowable values should be determined. It is often the case that the input space is very large and furthermore, for those situations where there are continuous inputs, the input space will be infinitely large. When faced with large input spaces, test engineers usually resort to a technique known as equivalence partitioning (Myers et al., 2004). The idea is to examine each input and, where necessary, divide the range of the input into a number of mutually exclusive categories known as equivalence classes. The expectation is that, from the standpoint of the software system, any individual member of an equivalence class is representative of the entire class. Also, as proposed in this paper, the test engineer should consider any blocking factors that may be appropriate and should also identify possible constraints on the input space that may be necessary.

Design: In the design phase, the combinatorial test suite is generated, and for each test case in the suite, the expected outcome must be determined. In order to generate the test suite, the test engineer first maps the inputs, input values, constraints, and the t-way combinations to be covered, that were identified during the design phase, to a covering array specification. Once this is done, the covering array is constructed, and the

resulting rows of the covering array become the individual test cases of the suite. The mechanics of how this is done will be determined by the tool used for constructing the covering array. Morgan (Morgan, 2018) provides a list of some of the current tools available for constructing combinatorial test suites. Given a test suite, determining the expected outcome for each test case is perhaps the most important of all testing activities. A test engineer must have some way of knowing the expected outcome of a test case in order to be able to assess if the outcome is a success or a failure. This is known as the test oracle problem (see Barr et al. (2014) for an in-depth discussion).

Test: The test phase involves the execution of each of the test cases and the recording of the outcomes. Based on the outcomes determined in the Design phase, a failure is recorded for any test case for which the actual outcome does not match the expected outcome. If the actual outcome matches the expected outcome then the test case is deemed a success.

Analyse: If any failures are recorded in the Test phase, the test engineer is faced with the challenge of identifying the failure-inducing combinations, a process known as fault localisation (Ghandehari et al., 2013). We consider this process to be akin to the analysis step that one would do after collecting data for a designed experiment and so we prefer to use the label ' Analyse' to describe this phase.

Given our proposed methodology, let us revisit the test engineer assignment scenario from the previous section to further develop the idea of blocking factors mentioned in the design phase. In that example, we could treat the test engineers as a blocking factor. From the traditional design of experiments perspective, observations within a block are thought to be more similar than between blocks. Given the same set of inputs, one would expect there to be little to no difference between blocks, particularly if the system is deterministic. However, in the case of test engineers, it turns out that there is considerable variability in how different test engineers approach testing. These differences are often because engineers are inclined to notice different issues while testing. For example, a particular test engineer may be more likely to notice numeric issues, while another might tend to focus more on user interface or graphical issues. In the case of the XGBoost example, different test engineers would probably choose different data, and this difference would likely further exacerbate the differences between test engineers.

The issue of assigning test engineers to testing components is related to assignment problems in Search Based Software Testing (SBST), that formulates testing as an optimisation problem (Harman et al., 2015). For automated bug assignment, Baysal et al. (2009) used developer predilections to assign bugs. While we could use test engineer strengths or preferences for assignment, for this paper we want to ensure that test cases for each test engineer are different and, are different in such a way that when aggregated, the overall set of test cases increases coverage. Hence, we treat test engineers as a blocking factor. The ability to ensure coverage for each test engineer will require only a simple adjustment to the combinatorial testing approach that still uses the underlying covering array construct. Furthermore, as will be discussed later, this approach also aids in fault localisation.

Note that in the context of testing software, blocks could also be different operating systems, different computers, or different days of the week. That is, differences in how a software system behaves may be attributable to the underlying operating systems or underlying hardware differences, or changes during development to the software system throughout the week may exhibit a day-of-the-week effect. When thought of in the context of typical designed experiments, these blocks can be fixed (test engineers) or random (day of the week). This does not affect the design, and the describe and design phases would also be unaffected. If the blocks are random, it is necessary to ensure that failures recorded during the test phase are reproducible so that the analyse phase can be effective.

3. Notation and preliminaries

Consider an array D with n rows and l columns. Let column i have v_i levels for $i = 1, \ldots, l$. D, denoted by $\mathbf{CA}(N, t, (v_1, \ldots, v_l))$, is said to be a covering array of strength t if any subset of t columns has the property that all $\prod v_i$ level combinations occur at least once. For those familiar with strength t orthogonal arrays, the definition is similar in that all combinations for a given subset of t columns must occur equally often.

By relaxing the 'equally often' restriction for an orthogonal array, the size of a covering array is always less than or equal to the size of the corresponding orthogonal array and, as we increase the number of columns, the number of levels, or the strength, the covering array size grows more slowly than the orthogonal array. To illustrate, let **CAN** and **OAN** denote the minimum run size of covering arrays and orthogonal arrays, respectively. If we consider the case where there are 35 two-level inputs, the **CAN** is 8 while the **OAN** is 36. If we increase the number of two-level inputs to 1000, the **OAN** increases to 1004 while the **CAN** increases to 14, less than twice the size of the case with 35 inputs.

The economical run size of covering arrays makes them an ideal construct to derive test cases for combinatorial testing. Since a strength t covering array ensures that all combinations of values for any t inputs are covered, a covering array-based combinatorial testing approach allows test engineers to be confident that software faults that can be precipitated by t or fewer inputs will be identified by the test suite. As pointed out earlier, these lower-order combinations are the most

likely to induce failures by the combination hierarchy principle. If instead, the test engineer generated a test suite 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. What is more, with random testing, a test engineer is not considering the coverage properties of the test suite, missing some combinations of a few inputs.

3.1. Addition of blocking factors

To add blocking factors for consideration in a test suite, we simply need to consider the blocking factors as additional factors/inputs in a covering array. For test engineers using covering arrays for testing, these additional factors are trivial to add in creating a test plan, but often not included since they may be outside of the traditional types of inputs typically considered. We assume in this paper that the blocking factors can vary independently. Our goal is to add blocking factors to a set of inputs, such that the set of test cases for each level of a blocking factor forms a strength t covering array. With the addition of blocking factors as additional inputs to the inputs of the covering array required for testing, a covering array of strength $t+1$ can be created, thereby increasing the overall coverage of the full test suite. If the blocking factor has v_1 levels, we know that the $t+1$ strength covering array will be at least v_1 times as large as the strength t covering array for the remaining inputs, not including the blocking factor (Sarkar & Colbourn, 2019). That is,

$$\begin{aligned}&\mathbf{CAN}(t+1,k,(v_1,v_2,\ldots,v_k))\\&\quad\geq v_1\mathbf{CAN}(t,k-1,(v_2,\ldots,v_k)).\end{aligned}$$

Furthermore, we know that we can construct a strength $t+1$ covering array which includes the blocking factor so that, for each level of the blocking factor, the remaining inputs constitute a strength t covering array. To see this, consider a strength $t+1$ covering array in k inputs. If we select any $t+1$ columns such that one of the columns is the blocking column, and consider the remaining t columns for any level of the blocking column, then those t columns must cover the values for those columns, by the definition of a covering array.

The set of test cases defined by the covering array then has the following properties:

(1) For each level of a blocking factor, the subset of test cases for that level will form a strength t covering array on the remaining inputs.
(2) The full test suite including the blocking factors is a strength $t+1$ covering array.

In this article, we only consider test suites for which the full design is strength $t+1$. While this requirement could be relaxed, being able to ascertain that there are no faults present due to combinations of $t+1$ inputs is a substantial improvement over combinations of t inputs. As long as the number of levels of any blocking factor is smaller than the t inputs with the largest number of levels, then the lower bound on the run size of the strength $t+1$ covering array is not typically influenced by the blocking factor. That is, there may not be a large change in the size of the test suite by considering a blocking factor unless the number of levels for blocks is large.

Example 3.1: Consider a test group of three test engineers tasked with testing a software system with nine binary inputs, A to I, each with two levels (i.e., 1 and 2). To divide the work, the engineers could decide to each focus on three of the nine inputs. However, for such a partitioning scheme any test engineer's test suite would likely miss any failure-inducing combinations that involve inputs not in their partition. To improve upon this setup, each test engineer could decide to test all nine inputs using a combinatorial testing approach. The **CAN** for a strength 2 covering array given 9 binary inputs is 6, so each test engineer's test suite would contain 6 test cases. However, this scheme treats the test engineers as independent entities and does not take into account the benefits that could accrue if the testing effort was treated as a combined effort where test engineers are treated as a blocking factor (i.e., an additional 3-level input where each level represents one of the test engineers). The strength 3 covering array in Table 1, is an array with 10 inputs where the first input represents the blocking factor and the remaining inputs are the nine binary inputs, A to I. By using the blocking approach, each test engineer's test suite would still contain 6 test cases and is still a strength 2 covering array but, by treating the testing task as a combined effort and pooling the results, the overall test suite on the inputs A to I form a strength 3 covering array.

Table 1. Blocked test suite for Example 3.1.

Test Engineer	A	B	C	D	E	F	G	H	I
1	1	1	2	1	1	2	2	1	2
1	2	2	1	1	2	2	2	1	1
1	2	1	2	2	2	1	1	1	2
1	2	1	1	2	1	1	2	2	1
1	1	2	1	1	2	1	1	2	2
1	1	2	2	2	1	2	1	2	1
2	1	2	1	2	1	1	2	1	2
2	1	1	1	2	2	2	1	1	1
2	1	1	2	1	2	1	2	2	1
2	2	2	2	1	1	1	1	1	1
2	2	1	1	1	1	2	1	2	2
2	2	2	2	2	2	2	2	2	2
3	1	1	1	2	2	2	2	2	2
3	1	1	1	1	1	1	1	1	1
3	1	2	2	1	2	2	1	1	2
3	2	1	2	2	1	2	1	2	1
3	2	2	2	1	1	1	2	2	2
3	2	2	1	2	2	1	2	1	1

Name	Role	Values					
max_depth	Categorical	3	9				
subsample	Categorical	0.3	1				
colsample_bytree	Categorical	0.3	1				
min_child_weight	Categorical	1	10				
alpha	Categorical	0	2				
lambda	Categorical	0	2				
learning_rate	Categorical	0.05	0.3				
iterations	Categorical	20	300				
tree_method	Categorical	auto	exact	approx	hist	gpu_exa	gpu_hist
predictor	Categorical	cpu_predictor	gpu_predictor				
grow_policy	Categorical	depthwise	lossguide				
booster	Categorical	gbtree	gblinear	dart			
process_type	Categorical	default	update				
sample_type	Categorical	uniform	weighted				
normalize_type	Categorical	tree	forest				
feature_selector	Categorical	cyclic	shuffle	greedy	thrifty		
colsample_bylevel	Categorical	0.3	1				
colsample_bynode	Categorical	0.3	1				
max_delta_step	Categorical	0	10				
gamma	Categorical	0	10				
scale_pos_weight	Categorical	1	10				
num_parallel_tree	Categorical	1	10				
base_score	Categorical	0.3	1				
nthread	Categorical	0	10				
seed	Categorical	0	12345				
sketch_eps	Categorical	0	.5				
refresh_leaf	Categorical	0	1				
max_leaves	Categorical	0	10				
max_bin	Categorical	256	512				
rate_drop	Categorical	0	.5				
one_drop	Categorical	0	.5				
skip_drop	Categorical	0	.5				
top_k	Categorical	0	10				
tweedie_variance_power	Categorical	1.5	2				

Figure 2. Inputs and the levels used for validating XGBoost.

4. Example using XGBoost

We now look at an example of our proposed method using XGBoost Release 0.90 (Chen & Guestrin, 2016). The objective is to derive a set of test cases to validate the hyperparameters of the library. In the describe phase, the test engineers identified 34 inputs. The continuous inputs in this example were discretised into categorical inputs of only two levels, in what is deemed as a low and a high value. The resulting list of inputs have one 6-level input, one 4-level input, one 3-level input, and thirty-one 2-level inputs. Figure 2 shows the inputs and associated levels. In actuality, there are some inputs that are only relevant when used in conjunction with a particular level of a different input. These can be taken into account in the design construction by specifying these disallowed sets of levels, also called forbidden edges (Danziger et al., 2009) or disallowed combinations (Morgan et al., 2017). For this example, we will assume that there are no constraints on the input space. Not only does this simplify the presentation here, but it is also useful in this case since it allows the test engineer to validate that specifying disallowed combinations to the software do not have unexpected side effects.

There is a pool of eight test engineers available, and it is desirable to weigh how many test engineers to assign to this testing effort while balancing their workload. To achieve a strength 2 covering array, the theoretical lower bound on the number of test cases is 24 (due to one 6-level and one 4-level input). Similarly, for a strength 3 covering array, the lower bound is 72. The test engineers want to be able to cover all combinations of levels for up to 3 inputs, but 72 test cases is determined to be too burdensome for each individual test engineer. Likewise, giving each test engineer a smaller subset of inputs will not ensure that 3-input combinations of the entire set of inputs will appear.

In this example, the test engineers are specified as a blocking factor in the describe phase. The goal is to provide each test engineer a set of test cases that is a covering array of strength 2, while the aggregated covering array for all test engineers is strength 3. For the design phase, we could consider varying the number of test engineers from two to eight. This is useful in practice to see if the resources (in this case test engineers) can be used elsewhere. Table 2 shows the average number of test cases per block for two to eight test engineers (blocks). Each of the designs were created using

Table 2. Average test cases per block and test suite size for XGBoost example in Section 4.

Blocks	Average test cases per block	Full test suite
2	36	72
3	26	78
4	24	96
5	24	120
6	24	144
7	24	168
8	24	192

the covering array implementation in JMP Software, with 10,000 iterations of a post-optimisation technique that tries to minimise the size of the covering array. We were able to create a design with no blocking factor that achieves the minimum run size for both strength 2 and 3. Based on these results, we can see that only for two and three blocks would individual test engineers require additional test cases beyond the minimum for a covering array of strength 2.

While the lower bound for a strength 4 covering array in this case is 144, the smallest strength 4 design we were able to construct had 266 runs. Interestingly, for the case with 8 blocks, while it does not achieve 100 percent 4-coverage (i.e, the percentage of 4-input combinations that appear in the test suite), it has 99.77% 4-coverage and 96.86% 5-coverage. Even for 4 blocks, the 4-coverage is 98.24% and the 5-coverage 87.77%. As a result, even if there exists a failure-inducing combination involving four inputs, it is likely covered by the test suite.

4.1. Expanded example

We now expand the XGBoost example, and consider the case where it is desirable to examine a subset of the inputs at additional levels. This may be useful for those inputs known to be more important to practitioners or for newly added inputs that have not been thoroughly tested. For example, let us say that in the describe phase, the test engineers want to examine 8 of the continuous inputs in greater detail. After partitioning these inputs as discussed in Section 2, four are at 5 levels and four at 4 levels. Those inputs and their new levels are given in Figure 3. In this setup, there is one 6-level input, four 5-level inputs, five 4-level inputs, one 3-level input, and twenty-three 2-level inputs. While the theoretical lower bound for a strength 2 covering array is 30 runs, the smallest unblocked design we constructed had 37 runs. For strength 3, the lower bound is 150 runs, while the smallest unblocked design we constructed had 244 runs. With the inputs and levels in this case, a substantially larger number of test cases are needed as can be seen in Table 3.

In the simpler example, in nearly every case, each block contained the minimum number of test cases for a strength 2 covering array. Increasing the number of blocks in this case increased the size of the test suite, thereby increasing the 4-coverage. In the more complex case, the increase in the size of the full test suite is smaller relative to the reduction in the number of test cases per block. The ease of adding a block to a set of inputs allows test engineers to explore different block sizes to decide the best allocation of their resources. By using the blocking approach to covering arrays, it is guaranteed that each test engineer is examining all possible two-way combinations of these important inputs.

5. Advantages of blocking in fault localisation

Our discussion to this point has focused on the design aspect of a test suite only. As mentioned earlier, one of the advantages of combinatorial testing with a covering array of strength t is that it provides confidence that a disciplined approach to testing is being applied and ensures that any failures due to t or fewer inputs will be induced. We now address the issue of fault localisation (Ghandehari et al., 2013), which takes place in the analyse step. That is, once a failure is observed, a

Table 3. Average test cases per block and test suite size for expanded XGBoost example in Section 4.1.

Blocks	Average test cases per block	Full test suite
2	143	246
3	82	246
4	64	256
5	58.2	291
6	48.5	291
7	44.14	309
8	39	312

Name	Role	Values				
max_depth	Categorical	3	5	7	9	
subsample	Categorical	0	0.3	.7	1	
colsample_bytree	Categorical	0	0.3	.7	1	
min_child_weight	Categorical	1	4	7	10	
alpha	Categorical	0	.5	1	1.5	2
lambda	Categorical	0	.5	1	1.5	2
learning_rate	Categorical	0	0.05	0.3	0.5	1
iterations	Categorical	20	100	200	300	500

Figure 3. Levels for expanded inputs for validating XGBoost.

test engineer wants to isolate the failure-inducing combination. Due to the run size economy of combinatorial testing, when faced with a failure, a test engineer may have a large list of potential causes (i.e., possible failure-inducing combinations) to investigate. As in the hierarchy principle for factorial effects, a test engineer wants to focus on the simplest explanations, the potential causes involving the fewest number of inputs. As an example, in one such realisation of a strength 2 covering array in the XGBoost example from Section 4, if a failure occurs in test case i and the combination of tree method set to auto and feature selector set to thrifty is the true cause, the test engineer would have to investigate eight potential two-way combinations. A test engineer then must go about determining which of those potential causes induces a failure.

In the case where test engineers is the blocking factor, if a failure is induced in the strength t subset of a particular test engineer, then the engineer knows that all other test engineers have those t-input combinations contained within their test suite. If the definition of failure is consistent among test engineers, then all test engineers should find the failure. For the types of failures that are known to be consistently checked, this implies that the results from all test engineers can be considered as a whole. Any failure-inducing combinations involving fewer than $t+1$ inputs will be easily induced and isolated and those due to at least $t+1$ inputs can be studied using fault localisation techniques.

There are failures that may not be obvious to some test engineers. In the XGBoost example, there may be a statistic that not every test engineer has checked. In this situation, the advantage of blocking is that each test engineer has tested all combinations of t inputs, so failures that rely on a test engineer's individual specialties will be covered. If a test engineer uncovers a failure that is not being tested among all test engineers, they can generate a set of potential causes based on their test suite. They can then coordinate with other test engineers to check for the failure condition in one or more test cases that contain the potential cause. In the authors' experience, for cohesive teams this is a painless experience - a test engineer will ask their colleagues to add an additional check in their test case(s) that contain one or more of the potential causes and check the results. This can speed up the fault localisation effort since, if it is due to a t-input combination, one of the other test engineers will induce the same failure. To reduce the burden of investigating all the different potential causes, it is advantageous to use any information about the failures and inputs to start with potential causes more likely to induce a failure (Lekivetz & Morgan, 2020). If no other test engineer observes the same failure, then it can be concluded that the failure is due to a combination involving $t+1$ or more inputs.

To further illustrate how blocking can aid fault localisation, consider the case where the block is workday, such as day of the week. For software development organisations, there may be a number of changes that are made to the code base daily. A test engineer may want to identify any failures introduced by newly submitted code as soon as possible. With time and resource constraints, there may be limits to the amount of test cases that can be performed on a given day, and a limit to the number of test cases to be created. Using the blocking approach in this paper with a strength $t+1$ covering array for the full test suite, if a test engineer observes a failure on a particular day, they know that for the previous day all t-input combinations were tested. If there were no failures the previous day (or any failures were deemed fixed), then the observed failure is most likely either due to a new failure-inducing combination involving less than $t+1$ factors, or a recently introduced fault that resulted in a failure that was induced by more than t inputs that has not been tested in the previous few days. To determine which situation it is, it is simply a matter of running the failure-inducing test on the previous day's version of the software. If the test still fails, then it is due to a failure-inducing combination from $t+1$ or more inputs and can be tracked back to the day it started to fail. If it does not fail the previous day, then the failure is due to a new change, and can be investigated using the hierarchy principle. In this case, a test engineer can combine test cases from versions of the software for days that exhibit the failure with the particular test to aid in fault localisation by utilising additional test cases that can be viewed as a success.

6. Discussion

In this paper, we have discussed how the concept of blocking in experimental designs can be applied to designing a test suite in a combinatorial testing framework. This involved presenting a combinatorial testing approach methodology that follows steps similar to those one applies in a designed experiment. The application of blocks is simple for test engineers already familiar with combinatorial testing techniques, as it is simply a matter of adding the blocking factor as an additional input to the covering array specification. The blocking approach also aids in fault localisation, where test engineers need to isolate failure-inducing combinations when failures are uncovered during testing. We assumed throughout that all blocks would involve testing all inputs. This approach still applies when some blocks only use certain subsets of inputs, by restricting combinations involving those blocks via disallowed combinations. One might use this approach to account for test engineer preferences or specialisations. We have also considered the case where the full test suite has strength $t+1$ and strength t for each level of a blocking factor. If there are enough blocks, one could make

the full test suite strength $t + m$, $m > 1$, with variable strength covering arrays, by which the software inputs would be specified to have a higher strength than subsets involving the blocking factors (Cohen et al., 2003). The requirement that each level of the blocking factor forms a strength t covering array is similar to that of sliced orthogonal arrays (Qian & Wu, 2009). Extending this idea to covering arrays would mean that the number of levels of an input in the full array could be more than the number of levels for each block. Each subarray would be a covering array based on fewer levels for some inputs.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Notes on contributors

Ryan Lekivetz is a Principal Research Statistician Developer for the JMP Division of SAS where he implements features in the Design of Experiments platforms for JMP software.

Joseph Morgan is a Principal Research Statistician/Developer in the JMP Division of SAS Institute Inc. where he implements features for the Design of Experiments platforms in JMP software. His research interests include combinatorial testing, empirical software engineering and algebraic design theory.

References

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2014). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, *41*(5), 507–525. https://doi.org/10.1109/TSE.2014.2372785

Baysal, O., Godfrey, M. W., & Cohen, R. (2009). A bug you like: A framework for automated assignment of bugs. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 297–298). IEEE.

Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). ACM.

Cohen, M. B., Dwyer, M. B., & Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (pp. 129–139). ACM.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., Colbourn, C. J., & Collofello, J. S. (2003). A variable strength interaction testing of components. In *Proceedings 27th Annual International Computer Software and Applications Conference. Compac 2003* (pp. 413–418). IEEE.

Dalal, S. R., & Mallows, C. L. (1998). Factor-covering designs for testing software. *Technometrics*, *40*(3), 234–243. https://doi.org/10.1080/00401706.1998.10485524

Danziger, P., Mendelsohn, E., Moura, L., & Stevens, B. (2009). Covering arrays avoiding forbidden edges. *Theoretical Computer Science*, *410*(52), 5403–5414. https://doi.org/10.1016/j.tcs.2009.07.057

Ghandehari, L. S., Lei, Y., Kung, D., Kacker, R., & Kuhn, R. (2013, November). Fault localization based on failure-inducing combinations. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 168–177). IEEE. https://doi.org/10.1109/ISSRE.2013.6698916

Harman, M., Jia, Y., & Zhang, Y. (2015). Achievements, open problems and challenges for search based software testing. In 2015 *IEEE 8th International Conference on Software Testing, Verification and Validation* (ICST) (pp. 1–12). IEEE.

Kuhn, D. R., Wallace, D. R., & Gallo, A. M. (2004, June). Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, *30*(6), 418–421. https://doi.org/10.1109/TSE.2004.24

Lekivetz, R., & Morgan, J. (2020). Covering arrays: Using prior information for construction, evaluation and to facilitate fault localization. *Journal of Statistical Theory and Practice*, *14*(1), 7. https://doi.org/10.1007/s42519-019-0075-2

Morgan, J. (2018). Combinatorial testing: An approach to systems and software testing based on covering arrays. *Analytic Methods in Systems and Software Testing*, 131–158.

Morgan, J., Lekivetz, R., & Donnelly, T. (2017). Covering arrays: Evaluating coverage and diversity in the presence of disallowed combinations. In *2017 IEEE 28th Annual Software Technology Conference (STC)* (pp. 1–4). IEEE.

Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). *The art of software testing* (Vol. 2). Wiley Online Library.

Qian, P. Z., & Wu, C. J. (2009). Sliced space-filling designs. *Biometrika*, *96*(4), 945–956. https://doi.org/10.1093/biomet/asp044

Sarkar, K., & Colbourn, C. J. (2019). Two-stage algorithms for covering array construction. *Journal of Combinatorial Designs*, *27*(8), 475–505. https://doi.org/10.1002/jcd.21927.issue-8