Moving Forward with Combinatorial Interaction Testing

Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam Porter, Gulsen Demiroz, and Ugur Koc

Abstract—Combinatorial interaction testing (CIT) is an efficient and effective method of detecting failures that are caused by the interactions of various system input parameters. In this paper, we discuss CIT, point out some of the difficulties of applying it in practice, and highlight some recent advances that have improved CIT's applicability to modern systems. We also provide a roadmap for future research and directions; one that we hope will lead to new CIT research and to higher quality testing of industrial systems.

Index Terms—Combinatorial interaction testing, covering arrays

1 Introduction

Modern software systems frequently embody hundreds or even thousands of configuration options. For example, a recent version of the Apache web server has 172 user-configurable options – 158 of these are binary, eight are ternary, four have four settings, one has five, and the last one has six. As a result, this system has 1.8×10^{55} unique configurations. The implications for fully testing such a system are clear. It can't be done. Even if it only took one second to test each configuration, the time needed to test all possible system configurations is longer than the Earth has existed. Equally astounding calculations emerge when looking at other kinds of system variabilities, that also require testing, such as user inputs, sequences of operations, or protocol options.

For this reason alone, the testing of industrial systems will almost always involve sampling enormous input spaces and testing representative instances of a system's behavior. In practice, therefore, this sampling is commonly performed with techniques collectively referred to as combinatorial interaction testing, (or CIT) [5], [7]. CIT typically models a system under test (SUT) as a set of factors (choice points or parameters), each of which takes its values from a particular domain. Based on this model, CIT then generates a sample, meeting a specified coverage criteria. That is, the sample contains some specified combinations of the factors and their values. For instance, pairwise testing requires that each possible combination of values, for each pair of factors, appears at least once in the sample. This is the most common case, and is typically realized through the use of a combinatorial structure called a covering array [7].

Techniques like CIT are currently being used in many domains, and a wide variety of free and commercial tools exist to support this process. We encourage readers interested in learning more broadly about this topic refer to a comprehensive survey, such as that of Nie and Leung [7]. Despite its many successes, it can be difficult to apply CIT in practice. Therefore, our goal in this paper is to focus on how researchers and practitioners are working to make it easier to apply CIT in practice. To do this, we will point out some of the practical difficulties of applying CIT, and discuss recent advances and open avenues for research that have come out of the application of CIT to modern systems. In particular, we believe that it's time to broaden CIT, moving beyond the traditional view of CIT as a static method that models and samples a system's user inputs or configurations, towards a new understanding in which CIT is used for alternative notions of input, such as sequences of events and software product lines, by examining more flexible notions of coverage, including varying the coverage requirements based on test outcomes or cost considerations, and by adopting incremental and adaptive approaches that don't treat testing as a batch job, but carry it out over multiple, feedback-driven iterations. Finally, we also provide our roadmap for future research and directions; one that should hopefully lead both to new CIT research and better testing of real systems.

At a high level, CIT can be broken down into four major phases, shown in Figure 1. The first two of these

- *Cemal Yilmaz, Gulsen Demiroz, and Ugur Koc are with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey.*
  E-mail: {cyilmaz, gulsend, ugurkoc}@sabanciuniv.edu
- *Sandro Fouché is with the Department of Computer and Information Sciences, Towson University, Towson, MD 21252.*
  E-mail: sfouche@towson.edu
- *Myra B. Cohen is with the Department of Computer Science and Engineering, University of Nebraska, Lincoln, NE 68558.*
  E-mail: myra@cse.unl.edu
- *Adam Porter is with the Department of Computer Science , University of Maryland, College Park, MD 20742.*
  E-mail: aporter@cs.umd.edu

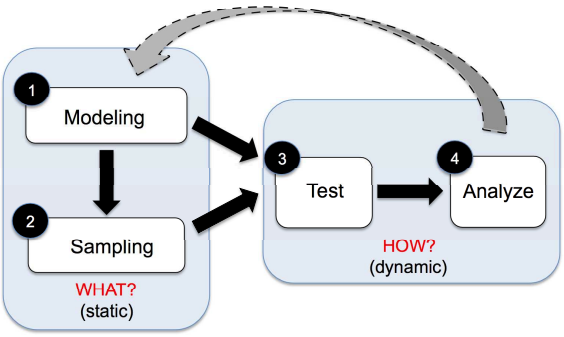

Fig. 1: Four Phases of CIT

phases, modeling and sampling, typically address the "WHAT" of testing – what are the characteristics of the SUT, and what are the inputs against which it should be tested? Modeling involves determining what aspect of the system to model (i.e., inputs, configurations, sequences of operations). Sampling refers to the process or algorithm by which we determine a means to cover the model generated in the first phase (e.g., all pairs of all factors, etc.). Currently, these phases are typically static, done once at the beginning of the process (though they can have optional feedback from the later phases).

The second two phases, testing and analysis, typically address the "HOW" of testing – actually running the tests and then examining the test results. These phases tend to be more process-driven than the first two phases, unfolding over a more extended period of time. In testing, developers may test in a batch mode, or test more incrementally or adaptively. And finally, developers analyze the test results, at a minimum to understand which test cases have passed and which have failed. In some cases, developers can use the testing and analysis phases to provide feedback to improve and refine later modeling and sampling activities.

2 Modeling

The first step of CIT is to model the SUT and its input space. The term *input* here is used in a general sense; anything that can affect the behavior of the system and that can be kept under control is considered an input.

The range of the entities that will be varied during testing logically defines the SUT's input space and is specified in the form of a model called the *input space model*. In this model, an input is normally expressed as a factor that takes on a small number of values. If the input factor is a continuous parameter or takes on a large number of values, then the factor will usually be discretized in some way, for instance, by using well-known techniques such as *equivalence partitioning* and *boundary value analysis*. Input space models can thus represent abstract or concrete tests.

A single set of values, one for each factor, is often referred to as a *configuration*. However, not all combinations of factor values may be valid as some factor values have dependencies. Such invalid combinations are often identified by logically specifying inter-factor *constraints*. For example, if factor 1 (`TCP/IP`) takes the value *true* then factor 2 (`NETWORK_ENABLED`) must take the *true* as well. An inter-factor constraint expressed in terms of factors and their values, thus invalidates some combinations of factor values, removing configurations containing those combinations from the *configuration space* – the set of all valid configurations.

Inter-factor constraints come in two varieties: system-wide constraints and test-case-specific constraints. System-wide constraints apply to any use of the SUT. Test-case-specific constraints, on the other hand, apply only to specific test cases run on the SUT, and are typically used to indicate input space configurations in which the test case cannot run. Constraints (either system-wide or test-case-specific) can also be classified as *hard* and *soft* constraints. Hard constraints mark the combinations of factor values that are not feasible or not permitted, whereas soft constraints mark combinations that are permitted, but undesirable, perhaps, because they are believed to provide little or no benefit during testing.

A model can also have a *seed*. A seed is a set of fully or partially specified set of combinations, which must or must not be part of any samples later drawn from this model. There are two common uses of the seeding mechanism: 1) to guarantee the inclusion of certain combinations or configurations in the sample and 2) to avoid testing already tested combinations.

3 Sampling

Input space models implicitly define the SUT's valid input space. CIT approaches systematically sample this input space, producing a set of configurations which will be tested in the next phase. The sampling is done by computing a highly economical combinatorial object, which by construction, satisfies a given sampling/coverage criteria. In this section we'll discuss various sampling criteria used by CIT approaches. Specifically, we'll focus on static sampling approaches, which generate a single sample of the input space.

3.1 Covering Arrays

A *t-way covering array*, for a given a input space model, is a set of configurations in which each valid combination of factor values for every combination of t factors appears *at least once* [7]. The parameter t is often referred to as the *coverage strength* and tools that construct these covering arrays will generally attempt

TABLE 1: An example traditional 2-way covering array.

A	B	C	D	E
0	1	1	2	0
0	0	0	0	0
0	0	0	1	1
1	1	1	0	1
0	1	0	0	2
1	0	1	1	0
1	1	1	1	2
1	0	0	2	1
1	0	0	2	2

to do so using the minimum number of configurations possible.

As an example, consider the following system with three binary factors: A, B, and C, each with possible values: 0 and 1, and two ternary factors: D and E, each with possible values: 0, 1, and 2. In the absence of any inter-factor constraints, this system has 72 valid configurations. A 2-way covering array for this system is shown in Table 1, which has 9 configurations. As promised, for any two factors, all possible pairs of factor values can be found in these 9 configurations.

The reason for using covering arrays is that they can cost-effectively exercise all system behaviors caused by the values of t or fewer factors. Furthermore, for a fixed strength t, as the number of factors increases, the covering array size represents an increasingly smaller proportion of the whole configuration space. Thus, very large configuration spaces can be efficiently covered.

In practice, several empirical studies suggest that low strength coverage tends to be correlated with high statement and branch coverage [7]. These studies also suggest that a majority of factor-related failures are caused by the interactions of only a small number of factors [7]. That is, in practice, t is much smaller than the number of factors, typically $2 \leq t \leq 6$ with t=2 being the most common case. Therefore, covering arrays can be an effective and efficient way of detecting faulty interactions – factors and their values that cause specific failures to manifest. Of course, once developers choose a particular value of t, if there are faulty behaviors involving more than t factors, t-way covering arrays may not detect them.

One problem that arises in practice with covering arrays is that of *masking effects*. If a configuration fails during testing, then none of the combinations of factor-values in that configuration can be considered covered; i.e. the failure of one combination masks the others. We also see masking effects when particular options are selected. For instance, if the `help` option is used in many programs, it simply shows the help menu and exits (leaving all other behavior untested).

3.2 Variable-Strength Covering Arrays

Covering arrays define a fixed strength, t, across all factors. However, it is sometimes desirable to test certain groups of factors more strongly (i.e., higher strength for certain factor groups) while maintaining a t-way coverage across the whole system. This is useful when, for example, it is expensive to increase t across all factors or when developers know that some factor groups are more likely to cause failures or cause more serious failures.

In essence, variable-strength covering arrays allow the coverage strength to vary across the configuration space. More formally, a *variable-strength covering array* is a covering array of strength t with subsets of factors of strength greater than t [7]. In the remainder of the document, fixed-strength and variable-strength covering arrays will be referred to as *traditional* covering arrays.

3.3 Error Locating Arrays

While traditional covering arrays help developers detect failures, static *error locating arrays* (ELAs) help developers detect and isolate faulty interactions [6]. ELAs do this by constructing covering arrays using a coverage criteria that builds systematic redundancy into the sample. Given certain assumptions, this redundancy allows the specific combination of factor values leading to a failure to be isolated.

ELAs, however, may not exist for all input space models. The exact conditions for the existence of ELAs can be found in [6]. One sufficient, but not necessary, condition for the existence of ELAs is that the input space model has *safe values* – at least one value for every factor in the input space model that is not present in any faulty interaction.

Given an input space model known to have safe values, a strength t, and an upper bound d on the number of faulty interactions, a (t, d)-way ELA is simply a traditional $(t + d)$-way covering array. If the input space model has no safe values, then a (t, d)-way ELA would be a traditional $t(d+1)$-way covering array.

3.4 Test Case-Aware Covering Arrays

In the earliest CIT efforts, input space factors corresponded to user inputs and, thus, each covering array configuration mapped to a single test case (made up of various user inputs). However, When CIT is applied to other kinds of inputs, such as system configuration options, test cases can become orthogonal to covering array configurations. Covering array configurations, in these cases, map to system configurations, and each test case in the SUT's test suite is run on each configuration in the covering array.

TABLE 2: An example 2-way test case-aware covering array.

A	B	C	D	E	scheduled test cases
0	1	1	2	0	$\{t_2, t_3\}$
0	0	0	0	0	$\{t_1, t_3\}$
0	0	0	1	1	$\{t_1, t_3\}$
1	1	1	0	1	$\{t_1, t_2, t_3\}$
0	1	0	0	2	$\{t_1, t_2, t_3\}$
1	0	1	1	0	$\{t_1, t_2, t_3\}$
1	1	1	1	2	$\{t_1, t_2, t_3\}$
1	0	0	2	1	$\{t_1, t_2, t_3\}$
1	0	0	2	2	$\{t_1, t_2, t_3\}$
1	1	1	2	0	$\{t_1\}$
0	1	0	2	1	$\{t_1\}$
1	0	0	0	0	$\{t_2\}$
0	1	0	1	1	$\{t_2\}$

Some test cases may be runnable in specific systems configurations, for instance, because the test case tests a feature that is disabled in certain configurations. In these cases, developers will need to include test-case-specific constraints in their initial input space model.

In the presence of unaccounted for test-case-specific constraints, traditional covering arrays are particularly vulnerable to masking effects. Specifically, when a test case fails to run in a particular configuration, nearly all of the factor combinations captured in that configuration have not actually been tested.

Test case-aware covering arrays address this problem by allowing developers to specify both system-wide constraints and test-case-specific constraints, and then taking these constraints into account when constructing covering arrays [11]. A t-way test case-aware covering array is constructed in such a way that, 1) none of the selected configurations violate system-wide constraints, 2) no test case is scheduled to be executed in a configuration that violates its test-case-specific constraint, and 3) for each test case, every valid t-way combination of factor values for the test case appears at least once in the set of configurations in which the test case is scheduled to be executed.

As an example, consider the SUT discussed in Table 1 which needs to be tested by using three test cases: t_1, t_2, and t_3. Test cases t_1 and t_2 have some test-case-specific constraints: t_1 cannot run when $(A{=}0 \wedge C{=}1)$, and t_2 cannot run when $(A{=}0 \wedge B{=}0)$. Test case t_3, on the other hand, has no test-case-specific constraints. Table 2 depicts a 2-way test case-aware covering array, for this scenario, which requires 28 test runs on a total of 13 configurations. As promised, all test-case-specific constraints are accounted for thus no masking effects caused by violated constraints occur.

In practice, there is often a trade-off between minimizing the number of configurations and minimizing the number of test runs in test case-aware covering arrays. Therefore, naive techniques, such as creating a traditional covering array for each test case in isolation, may result in overly large covering arrays [11].

3.5 Cost-Aware Covering Arrays

The sampling criteria we have discussed so far, assume a simple execution cost model – in which every configuration (or test run) has the same cost. If tests have such a uniform cost, then you can reduce the cost of testing by reducing the number of configurations (or test runs) required. This model, however, does not fit well with all test scenarios. For example, some configurations can cost more to construct then others – such as ones that require software installation or compilation. With configuration-dependent costs, reducing the number of configurations or test runs does not necessarily reduce the total cost of testing [2], [9].

Cost-aware covering arrays take the estimated actual cost of testing into account, when constructing interaction test suites. In particular, a t-way cost-aware covering array is a traditional t-way covering array that "minimizes" a given cost function. One approach to compute such covering arrays is to take as input a precomputed covering array and reorder the selected configurations such that the switching overhead is minimized [9]. Another approach is to consider the cost directly when building the covering arrays [2]. For example, in an SUT with runtime factors A and B and compile-time factors C, D, and E, changing the settings for factors C, D, and E will require a partial or full rebuild of the system – which is costly. On the other hand, factors A and B are set at runtime and the cost of doing so is negligible compared to that of building the system. In this scenario, reducing the cost of testing is the same as reducing the number of times the system is built, i.e., the number of compile-time configurations. In [9] the difference in runtime order could be on the order of weeks, when the cost of installing new software between configurations is considered.

3.6 Sequence-Covering Arrays

With traditional covering arrays, the order of factor values in a given configuration is assumed to have no effect on the fault revealing ability of the configuration. Any permutation of the factor values present in a configuration covers the same set of factor value combinations, and should detect the same faulty interactions. This assumption, however, does not hold in event-driven systems, such as found in graphical user interfaces and device drivers, where the way an event is processed often depends on the sequence of preceding events. Therefore, different orderings of the same set of events can reveal different failures.

Sequence-covering arrays are built to cover orderings of events that are implicitly specified by a given coverage criterion, in a "minimum" number of fixed-length event sequences [5], [12]. Existing approaches differ

in the coverage criteria they employ. One criterion, for example, ensures that every possible sequence of unique events of length t is tested at least once, while the events in the sequence can be interleaved with other events [5]. Another criterion ensures that every possible permutation of t consecutive events starting at every possible position in a fixed-length event sequence, is tested at least once [12]. Sequence-covering arrays have been so far used for testing graphical user interfaces [12] and testing a factory automation system [5].

All told, CIT sampling approaches take an input space model and generate the smallest set of configurations they can find that meet a specified coverage criteria. Different approaches vary in terms of the models they use and the criteria they attempt to cover.

4 Testing

When implementing a CIT process practitioners need to 1) decide on key CIT parameters (t, input space model, constraints, etc.), 2) execute test cases, and 3) analyze the resulting test data, for instance, to isolate any observed faults. Traditional CIT approaches have required developers to determine these parameters up-front, and then to execute and analyze tests as a one-shot, batch process. Each CIT step, however, poses significant technical challenges which are complicated by the dynamic and inherently unpredictable nature of testing.

Traditionally, developer judgement has guided the first step of deciding key parameters. Developers have had to guess at the right sampling strength (t), create their own input space models, and determine any relevant constraints. This is always tricky because there are no concrete rules on which developers can reliably base these judgements. In addition, each of these key parameters varies, not only with the characteristics of SUT, but also with an SUT's lifecycle stage, economic constraints, and more. For example, as the SUT evolves, the input space model may need to be revised to reflect areas of code churn, newly discovered constraints, or emerging masking effects. Depending on the size of the input space and the sampling methodology used, scheduling test case management and execution can become complicated. Test processes may require sophisticated support for build and test execution and for test case prioritization. Finally, masking effects can make it difficult to isolate the factors responsible for specific test failures.

To address these problems, researchers have begun to focus on new CIT approaches, that try to relieve developers of the need to make so many static, up-front parameter decisions. A key strategy behind this research has been to make CIT incremental and adaptive, so that decisions can be made dynamically based on the observable behavior of the SUT. Such adaptation is used, for instance, to establish key test parameters, to learn the SUT's input space model, and to react to failures that may create masking effects.

4.1 Determining Key Values

Typically developers base input space models and constraints on their knowledge of the SUT. They also use their judgement to determine the desired sampling strength – t. Unfortunately, there are few, if any, reliable guidelines for determining appropriate values for these key parameters and the result of choosing incorrectly can be either under-testing the system, leaving out key factors, failing to consider key constraints, or over-testing at significant cost in time and resources.

In choosing a sampling strength, t, developers never know with any certainty what value will be needed to find and classify failures in a given system. If they choose pairwise interactions ($t = 2$) then any 3 or 4-way failures in the system may be not be found. If they choose strength 4, then 4-way and lower level failures will be correctly classified, but given the large size of the 4-way covering arrays, many configurations may have been unnecessary and any actual 2-way failures may not be found until most or all of the entire 4-way schedule is completed, delaying feedback to the SUT's developers.

Incremental Covering Arrays. Incremental covering arrays (ICA) address this problem by never choosing t at all, but rather by incrementally increasing sampling strength as testing resources allow [4]. ICA constructs each covering array using a *seed* taken from already run lower strength arrays, such that their size will be approximately that of a traditionally built covering array. As mentioned earlier, seeding means that we *fix* a set of configurations at the start, and construct the new covering array by filling in the required t-way interactions not already contained in the seed. Because an incrementally constructed t strength covering array is built using a previously constructed $(t-1)$-way array as a seed, the existing configurations are reused and only a smaller number of new configurations have to be run to get complete t-way coverage; i.e. classification of 2-way failures completes *before* embarking on 3-way coverage, but it costs almost the same as executing a traditional 3-way array.

Creating the input space model to begin with also relies on the incomplete and possible error-prone knowledge of software interactions by the development team. Incorrect models will result in confusing test errors, masking effects, and wasted testing effort. In addition, the assumption that the models map to real control or data dependencies in the code may not hold, leading to gaps and redundancies in testing.

Interaction tree discovery. Interaction tree discovery (iTree) removes the need to create an input space model by iteratively computing the effective input space using machine learning techniques [8]. iTree

works by instrumenting code coverage on the system under test, then computing a small set of configurations on which to test the SUT. After subsequent test execution, iTree uses the resulting coverage data as input to machine learning algorithms. These ML algorithms attempt to uncover conjunctions of factors and values that alter code coverage. iTree then iterates, computing new configuration sets that may increase code coverage. iTrees handles determining not only the input space model, but avoids the entire issue of choosing sampling strength by providing a type of variable strength sampling as a side-effect of the adaptive process.

4.2 Executing Tests

Managing the test process itself is also not a trivial problem. Traditionally, CIT research has not focused extensively on test case management and execution, nor has it fully considered the test case execution orders.

Test Execution Support Systems. Executing test schedules requires converting CIT covering arrays into runtime test execution, and requires processing the results to drive adaptive CIT. Several systems have been created to simplify this process. For example, the Skoll framework provides mechanisms that support CIT using covering arrays and performs continuous, distributed testing. Fouche et al [4] describe how Skoll was used to support a large-scale CIT process running on dozens of testing nodes, over a several month period, to sample an input space of 72+ million configurations for MySQL.

Prioritization. While CIT can successfully sample a large input space to test a single version of a software system, time and budget is always a consideration. When the amount of work required exceeds available time and resource constraints, developers will often want to run the most important tests first. Several approaches for *prioritizing*, or ordering CIT test suites exist. The goal of prioritization is to include important configurations early in testing, to maximize early fault finding. To address prioritization, Bryce and Colbourn [1] define a new type of covering array called a *biased covering arrays*. Biased covering arrays use a set of weights, provided by the tester, to cover the most important parameter values early in testing. Empirical studies show that using both biased covering arrays (as well as traditional covering arrays that are re-ordered) can improve early fault detection. The weights in this case are informed by code (or fault) coverage from earlier versions of the system. Another way to prioritize test suites (when information from prior versions is not available) is to simply prioritize by the number of interactions that have been covered; this is a biased covering array with equal weights on all values and has also been shown to be effective.

4.3 Reducing Masking Effects

When performing CIT, if one configuration fails to run to completion, then all the combinations of parameter values in that configuration are no longer guaranteed to have been tested. This leads to masking effects. Since faults may not be fixed immediately, or the failures may be due to improper modeling, adaptive methods have been proposed as a way to handle this problem.

For example, the iTree approach described earlier iteratively generates the input space model, and the process utilizes feedback from prior test executions to adapt the effective model. As an additional benefit, masking effects that alter the runtime code coverage are automatically addressed by the machine learning process. But this technique does not address masking effects which exercise the same sections of the source code, for instance by running the same section of code a different number of times or in a different order.

Feedback Driven Adaptive Combinatorial Interaction Testing (FDA-CIT). FDA-CIT [3] is designed specifically to address this problem. It first defines an interaction coverage criterion that aims to ensure that each test case has a fair chance to execute all of its required combinations of factor values. This criterion is then used to direct a feedback-driven adaptive process. Each iteration of this process detects potential masking effects, isolates likely causes, and then generates configurations that omit those causes, but still contain all the previously masked combinations. The process iterates until the coverage criterion has been achieved.

Adaptive Error Locating Arrays. Adaptive ELAs [6] is another approach that can handle masking effects. Adaptive ELAs look for conclusive evidence of masking effects, rather than relying on statistical evidence as FDA-CIT does. This approach works when certain strong assumptions are met. For example, one type of adaptive ELAs is defined only for $t = 2$ and requires that all faulty interactions involve at most two factors and that safe values (known not to be part of masking effects) are already known. Another type of adaptive ELAs, which does not require safe values to be known a priori, requires that all factors are binary. While not appropriate for every system, if these conditions hold, then adaptive ELAs are guaranteed to remove all masking effects.

5 Analysis

After testing, developers normally examine the test results. One of the first questions they ask is whether the tests passed or failed. When some test cases fail, developers will normally analyze the test results to better understand the observed failures and to search for clues on how to fix the underlying faults. Because covering arrays have a known structure, sophisticated

analyses can be perfomed. These analyses often involve identifying the factors and values that cause observed failures to manifest, which can help developers reduce the turnaround time for bug fixes. We call this process, *fault characterization*.

At a high level, there are two types of fault characterization approaches; *probabilistic* [3] and *exact* [6] approaches. With probabilistic approaches, fault characterization is done by testing a covering array and feeding the results to some kind of data mining algorithm, such as a classification tree algorithm. The output is a model describing the factors and their values that best predict the observed failures. The adaptive FDA-CIT process discussed in Section 4, for example, uses a probabilistic approach. Exact approaches [6], on the other hand, look for conclusive, rather than statistical, evidence in fault characterization, i.e., every potential failure diagnosis is validated by further testing, rather than being accepted once a statistical threshold has been reached. With these approaches, if a configuration in a covering array fails, that configuration is first divided into smaller partitions using a divide-and-conquer approach. Each partition contains a subset of the factor values present in the original failing configuration, with the remaining values replaced by safe values. These partitions are then retested to determine whether they are also failure inducing. The iterations terminate when all the faulty interactions in the original failing configuration have been determined.

Each of these fault characterization approaches have their own pros and cons. In particular, probabilistic approaches typically require fewer configurations, but can produce inaccurate results, whereas exact approaches produce accurate fault characterization models, but they typically do so at the cost of testing more configurations and requiring some a priori knowledge, such as the safe values. Therefore, the choice of approach depends on the tester's objectives and requirements.

A final question that developers may ask is whether each covering array configuration and test case provides unique testing value. For example, analyzing data from our previous work with the MySQL project we observed that clusters of test cases had perfectly correlated outcomes – one passed, they all passed; one failed, they all failed. Such information could be used to prune some test cases from our test suite, or, alternatively, to prioritize each test case's execution. We also observed patterns suggesting that some parameters were effectively "dead" with respect to some test cases; changing the value of the dead parameter, did not change the observed behavior of the test case. This kind of information could greatly limit the number of configurations that need to be tested, and can be obtained via analysis of the covering array configurations and their test results.

6 Research Roadmap

In this article, we have examined the theory and practice of CIT. We also discussed problems with the current CIT practice and provided an overview of some of the interesting efforts to overcome them. Some of our major findings and future directions by phase are:

Modeling. Several recent advances in CIT are driven by the idea that we can exploit the benefits of CIT on many "non-traditional" combinatorial spaces. In addition to traditional user inputs, CIT is increasingly being applied to different kinds of inputs, such as configuration options, GUI events, protocols, software product line features and web navigation events [5], [10], [12]. However, currently input space models are for the most part created manually, which is a cumbersome and error-prone process, causing testing to be incomplete or needlessly expensive. A challenge for the future is deriving partially or fully automated processes to extract the CIT model and to handle evolving models as they change over time. Such processes could (1) extract factors and their values from software artifacts, (2) suggest the constraints among factors, and (3) aid in deciding the interaction strength to which different groups of factors should be tested.

Sampling. Once the input space model is created, the space then needs to be sampled. Numerous sampling approaches exist, yet observe that many practical test scenarios are still poorly supported. We believe that the applicability of CIT approaches in practice would greatly be improved, if there were better tools that allowed practitioners to define their own application-specific models and coverage criteria. That is, rather than having researchers develop specific models and coverage criteria, provide a general way for practitioners to flexibly define their own criteria, supported by powerful tools for generating samples. Although such generic tools may not be as efficient as their specialized counterparts, they certainly can provide the flexibility needed in practice.

Test. No matter how accurate the initial model and sampling are, unanticipated events during testing, such as masking effects, or insufficient time can prevent CIT from achieving its objectives. Therefore, we believe that incremental and adaptive approaches that steer the test process to avoid consequences of unanticipated events, such that testing objectives can be achieved in a cost-effective manner, are key aspects of practical CIT.

Analyze. After (or during) testing, test results need to be analyzed. In this paper we focused only on fault characterization. One interesting avenue for future research is to combine probabilistic and exact fault characterization approaches to develop a hybrid fault characterization approach, which can reduce the testing cost, compared to exact approaches, yet improve accuracy, compared to probabilistic approaches. In

general better tool support for detecting and locating faulty interactions as well as assessing the thoroughness of interaction testing are of practical importance.

REFERENCES

[1] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006.

[2] G. Demiroz and C. Yilmaz. Cost-aware combinatorial interaction testing. In *Proceedings of the 2012 International Conference on Advances in System Testing and Validation Lifecycle*, pages 9–16, 2012.

[3] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 243–253, 2011.

[4] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188, 2009.

[5] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei. Combinatorial methods for event sequence testing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 601–609, 2012.

[6] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM Journal of Discrete Mathematics*, 23(4):1776–1799, 2009.

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

[8] C. Song, A. Porter, and J. S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 903–913, 2012.

[9] H. Srikanth, M. B. Cohen, and X. Qu. Reducing field failures in system configurable software: cost-based prioritization. In *Proceedings of the 20th IEEE international conference on software reliability engineering*, pages 61–70, 2009.

[10] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *IEEE International Conference on Software Maintenance*, pages 211–220, 2009.

[11] C. Yilmaz. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on*, 39(5):684–706, 2013.

[12] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE Trans. Softw. Eng.*, 37(4):559–574, 2011.