Moving Forward with Combinatorial Interaction Testing

Cemal Yilmaz, *Sabanci University, Istanbul, Turkey*
Sandro Fouché, *Towson University*
Myra B. Cohen, *University of Nebraska–Lincoln*
Adam Porter, *University of Maryland at College Park*
Gulsen Demiroz and Ugur Koc, *Sabanci University, Istanbul, Turkey*

Combinatorial interaction testing (CIT) is an effective failure detection method for many types of software systems. This review discusses the current approaches CIT uses in detecting parameter interactions, the difficulties of applying it in practice, recent advances, and opportunities for future research.

Modern software systems frequently offer hundreds or even thousands of configuration options. A recent version of the Apache webserver, for example, 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. Consequently, this system has 1.8×10^{55} unique configurations. The implications for fully testing such a system are clear: it cannot 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 variability that also require testing, such as user inputs, sequences of operations, or protocol options.

For this reason, the testing of industrial systems will almost always involve sampling enormous input spaces and testing representative instances of a system's behavior. In practice, this sampling is commonly performed with techniques collectively referred to as combinatorial interaction testing (CIT).[1,2] 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 that meets specified coverage criteria; that is, the sample contains 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.[2]

0018-9162/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

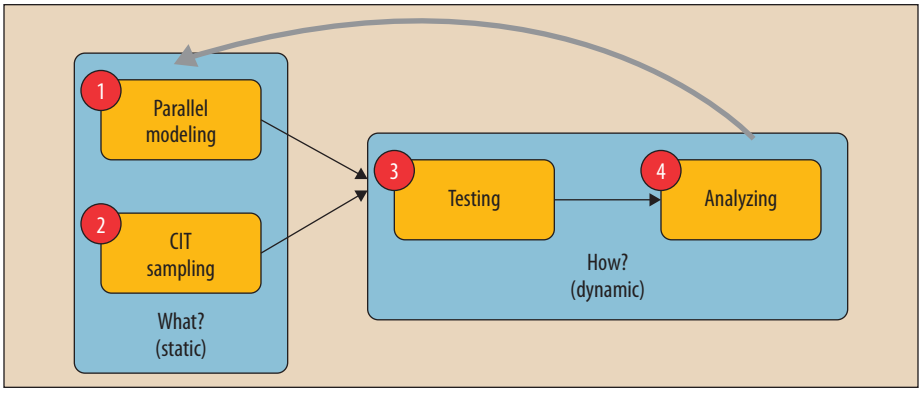

Figure 1. Four phases of combinatorial interaction testing (CIT). Phases 1 and 2 determine "what" is going to be tested, phases 3 and 4 address the "how."

Techniques such as CIT are currently used in many domains, and a wide variety of free and commercial tools exist to support this testing process. However, despite its many successes, CIT can be difficult to apply in practice, and researchers and practitioners are actively working to address this. Although there are practical difficulties in applying CIT to modern systems, recent advances and current work by researchers and practitioners show that CIT has the potential to go well beyond the traditional view that it is a static method that models and samples a system's user inputs or configurations. Through future research and development, CIT will continue to advance and lead to better testing of real systems.

At a high level, we can break down CIT into four major phases, as shown in Figure 1. The first two 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, such as inputs, configurations, and 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—for example, all pairs of all factors, and so on. Currently, these phases are typically static, done once at the beginning of the process (although they can have optional feedback from the later phases).

The second two phases, testing and analyzing, typically address the *how* of testing—actually running the tests and then examining the test results. These phases tend to be more process driven, unfolding over a more extended period of time. In testing, developers can test in a batch mode, incrementally, or adaptively. Ultimately, developers analyze test results to understand, at a minimum, 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.

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 system behavior and that can be kept under control is considered an input.

The range of entities that will be varied during testing logically defines the SUT's input space and is specified in the form of an 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 are 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 *true* as well. An inter-factor constraint is expressed in terms of factors and their values; thus, it 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 and test case–specific. 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 permitted. For instance, if the `NETWORK_ENABLED` feature is *false*, then the `REMOTE_PRINTING` feature cannot be *true*. Soft constraints, on the other hand, mark combinations that are permitted, but undesirable, perhaps because they are believed to provide little or no benefit during testing. For instance, `NETWORK_ENABLED` is *true* and `LOCAL_PRINTING` is *true* might be a valid combination, but it might not be of interest to test because local printing will not use the network.

A model might also specify a seed, which is a set of combinations that must or must not be part of any samples later drawn from this model. There are two common uses

of seeding: first, to guarantee the inclusion of certain combinations or configurations in the sample; and second, to avoid testing already tested combinations.

SAMPLING

The SUT's valid input space is implicitly defined by its input space model. CIT approaches systematically sample this input space, producing a set of configurations that will be tested in the next phase. The sampling is done by computing an efficient combinatorial object called a covering array, which, by its construction, satisfies a given sampling/coverage criteria.

Covering arrays

A *t-way covering array* for a given 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.[2] The parameter t is often referred to as the *coverage strength*, and tools that construct these covering arrays will generally attempt 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 two-way covering array for this system is shown in Figure 2, which shows nine configurations. As indicated, for any two factors, all possible pairs of factor values are found in these nine configurations.

Covering arrays could 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.[2] These studies also suggest that a majority of factor-related failures are caused by the interactions of only a small number of factors.[2] That is, in practice, t is much smaller than the number of factors, typically $2 \le t \le 6$, with $t = 2$ being the most typical case. Therefore, covering arrays can be an effective and efficient way of detecting faulty interactions—combinations of factors and their values that cause specific failures to manifest. Of course, if there are faulty behaviors involving more than t factors, t-way covering arrays might not detect them.

Masking effects are a kind of problem that can arise in practice with covering arrays. If a configuration manifests a failure during testing, then none of the combinations of factor-values in that configuration can be considered covered with high certainty; in other words, the failure caused by one combination could prevent other behaviors caused by other combinations from being exercised. We also see masking effects when particular options are selected. For instance, a `help` option is included in many programs, which simply shows the help menu and exits (leaving all other behavior untested).

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

Figure 2. A traditional two-way covering array for a system with 72 unique configurations requiring only nine configurations.

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—for example, higher strength for certain factor groups—while maintaining a t-way coverage across the whole system. This is useful in cases where 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. Put another way, a variable-strength covering array is a covering array of strength t with subsets of factors of strength greater than t.[2] We refer to fixed- and variable-strength covering arrays as traditional covering arrays.

Many commercial and free CIT tools are available for computing traditional covering arrays (see www.pairwise.org). In general, these tools differ from one another in

- their choice of algorithm for computing covering arrays;
- their effectiveness in reducing the size of covering arrays;
- their performance and scalability;
- the types of input spaces they support; and
- the features they have, such as supporting mixed-level factors, allowing the coverage strength to vary across the factors, providing a seeding mechanism, and handling inter-factor constraints.

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	$\{t_2\}$

Figure 3. An example two-way test case–aware covering array handling all known test case–specific constraints.

Error-locating arrays

Although traditional covering arrays help developers detect failures, static error-locating arrays (ELAs) help developers detect and isolate faulty interactions.[3] They 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, might not exist for all input space models. Conrado Martinez and his colleagues describe the exact conditions for the existence of ELAs,[3] and one sufficient—though not required—condition is that the input space model has *safe values*; that is, it has 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.

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, which was made up of various user inputs. However, when CIT is applied to other kinds of inputs, such as system configuration options, this is no longer true. Covering array configurations are not themselves different test cases: they are system configuration parameters under which the test cases in the actual test suite are run. An entire testing session thus involves running each test case in the SUT's test suite on each configuration in the covering array. Some test cases can only be run in specific system configurations—such as when testing a feature that is disabled in certain configurations. In these cases, developers need to include test case–specific constraints in their initial input space model.

In the presence of 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 will not have been tested.

Test case–aware covering arrays address this problem by allowing developers to specify both system-wide constraints and test case–specific constraints when constructing covering arrays.[4] Our experiences with highly configurable software systems suggest that, although test case–specific constraints are often encoded in test oracles,[4] therefore indicating that they are typically known by developers, manually determining and expressing all such constraints can still be a challenging task. A t-way test case–aware covering array is constructed in such a way that

- none of the selected configurations violate system-wide constraints;
- no test case is scheduled to be executed in a configuration that violates its test case–specific constraint; and
- 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 in Figure 2, which shows 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. Figure 3 depicts a two-way test case–aware covering array for this scenario that requires 28 test runs on a total of 13 configurations. As indicated, all test case–specific constraints are accounted for, so 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, could result in overly large covering arrays.[4]

Cost-aware covering arrays

For the sampling criteria we have discussed so far, we assume a simple execution cost model, in which every configuration (or test run) has the same cost. If tests have uniform cost, then the cost of testing is reduced by lowering the number of configurations (or test runs) required. This model, however, does not fit well with all test scenarios. For example, some configurations 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.[5,6]

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 this type of covering array is to take as input a precomputed covering array and reorder the selected configurations such that the switching overhead is minimized.[5] Another approach is to consider the cost when building the covering arrays.[6] 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, so the cost of changing these settings is negligible compared to that of building the system. In this scenario, reducing the cost of testing is effectively the same as reducing the number of times the system is built, specifically, the number of compile-time configurations.

Sequence-covering arrays

With traditional covering arrays, the order of factor-values in a given configuration is assumed to have no effect on the configuration's fault-revealing ability. 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 like those found in GUIs and device drivers, where the sequence of preceding events determines the way an event is processed. Therefore, differing sequences of the same set of events reveals different failures.

Sequence-covering arrays are built to cover sequences of events that are implicitly specified by a given coverage criterion in a "minimum" number of fixed-length event sequences.[1,7] Existing approaches differ in the coverage criteria they use. One criterion, for example, ensures that every possible sequence of unique events of length *t* is tested at least once, whereas the events in the sequence can be interleaved with other events.[1] 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.[7] Sequence-covering arrays have thus far been used for testing GUIs[7] and a factory automation system.[1]

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.

TESTING

When implementing a CIT process, practitioners need to decide on key CIT parameters (*t*, input space model, constraints, and so on), execute test cases, and analyze the resulting test data to isolate any observed faults. Traditional CIT approaches 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 complicated by the dynamic and inherently unpredictable nature of testing.

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.

Traditionally, developer judgment has guided the selection of key parameters. Developers tend to guess at the right sampling strength (*t*), create their own input space models, and determine any relevant constraints. This is tricky because there are no concrete rules on which developers can reliably base these judgments. In addition, each of these key parameters varies, not only with the characteristics of an SUT, but also with an SUT's life-cycle stage, economic constraints, and more. For example, as an SUT evolves, the input space model might 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 might 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 observable SUT behavior. 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.

Determining key values

Typically, developers base input space models and constraints on their knowledge of the SUT. They also use their judgment 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 include under-testing the system, leaving out key factors, failing to consider key constraints, or over-testing resulting in significant costs in time and resources.

In choosing a sampling strength t, developers do not 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 three- or four-way failures in the system might not be found. If they choose strength $t = 4$, then four-way and lower-level failures will be correctly classified, but given the large size of the four-way covering arrays, many configurations might have been unnecessary and any actual two-way failures might not be found until most or all of the entire four-way schedule is completed, thereby delaying feedback to the SUT's developers.

The goal of prioritization is to include important configurations early in the testing process, to maximize early fault detection.

Incremental covering arrays. Incremental covering arrays (ICAs) address this problem by never choosing t at all. Several related approaches exist.[8,9] In the approach described by Sandro Fouché, Myra Cohen, and Adam Porter,[8] for example, increases to sampling strength are made incrementally as testing resources allow. This technique constructs each covering array using a seed taken from already-run lower-strength arrays, so that their size will be approximately that of a traditionally built covering array. As mentioned earlier, seeding means fixing a set of configurations at the start, and then constructing a new covering array by filling in the required t-way interactions not already contained in the seed. Because an incremental t-way 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. In this approach, classification of two-way failures completes before embarking on three-way coverage, but the cost is almost the same as executing a traditional three-way array.

Creating the input space model 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 might 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 (ML) techniques.[10] iTree works by instrumenting code coverage on the SUT, 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 ML algorithms, which attempt to uncover conjunctions of factors and values that alter code coverage. iTree then iterates, computing new configuration sets that might increase code coverage. iTree handles determination of the input space model, but does not choose sampling strength; rather, it provides a type of variable strength sampling as an offshoot of the adaptive process.

Executing tests

Managing the test process itself is a nontrivial 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 test suites into runtime test execution, and then 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. Previous work[8] describes how Skoll was used to support a large-scale CIT process running on dozens of testing nodes, over several months, to sample an input space of more than 72 million configurations for MySQL.

Prioritization. Although CIT can successfully sample a large input space to test a single version of a software system, time and budget are always important considerations. 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 CIT test suites exist. The goal of prioritization is to include important configurations early in the testing process, to maximize early fault detection. To address prioritization, Renée Bryce and Charles Colbourn[11] defined a new type of covering array called a biased covering array. Such 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.

Reducing masking effects

During 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. Because faults might not be fixed immediately, or the failures might 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 ML process. However, this technique does not address masking effects that 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. Feedback-driven adaptive CIT (FDA-CIT)[12] is specifically designed to address the masking effect problem described earlier. To do this, it first defines the interaction coverage criterion that ensures that each test case has a fair chance to execute all of its required combinations of factor-values. Then, the criterion is 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[3] are another way to handle masking effects. They 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 ELA 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 ELA that does not require safe values to be known a priori requires that all factors are binary. Although not appropriate for every system, adaptive ELAs are guaranteed to remove all masking effects if these conditions hold.

ANALYZING

After testing, developers normally examine the test results, and one of the first questions they ask is whether the test cases 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 address underlying faults. Because covering arrays have a known structure, sophisticated analyses can be performed. These analyses often involve identifying the factors and values that caused observed failures, which helps reduce bug-fixing turnaround times. We call this process *fault characterization*.

At a high level, there are two types of fault-characterization approaches: probabilistic[12] and exact.[3] 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 values that best predict observed failures. The adaptive FDA-CIT process, for example, uses a probabilistic approach. On the other hand, exact approaches[3] look for conclusive—not statistical—evidence in fault characterization: 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 also induce a failure. The iterations terminate when all the faulty interactions in the original failing configuration have been determined.

The applicability of CIT approaches in practice would be greatly improved if there were better tools allowing practitioners to define their own application-specific models and coverage criteria.

These fault-characterization approaches each 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 typically 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.

Developers might also consider whether each covering array configuration and test case provides unique testing value. For example, in analyzing data from previous work[8] with the MySQL project, we observed that clusters of test cases had perfectly correlated outcomes—if one passed, they all passed; if 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.

Although this is not meant as a complete survey on CIT, we have explored some key problems with the current CIT practice and provided an overview of some of the interesting efforts to overcome them. The following paragraphs explore some important future directions for research listed by phase—modeling, sampling, testing, and analyzing, respectively. We encourage anyone interested in software testing to explore other novel directions in CIT as well.

Several recent modeling advances in CIT are driven by the idea that we can exploit the benefits of CIT on many "nontraditional" combinatorial spaces. In addition to traditional user inputs, CIT is increasingly applied to different kinds of inputs, such as configuration options, GUI events, protocols, software product-line features, and Web navigation events.[1,7] However, input space models are typically created manually today—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 extract factors and their values from software artifacts, suggest the constraints among factors, and aid in deciding the interaction strength for which different groups of factors should be tested.

Once the input space model is created, the space then needs to be sampled. Numerous sampling approaches exist, yet many practical test scenarios are still poorly supported. We believe that the applicability of CIT approaches in practice would be greatly improved if there were better tools allowing 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 might not be as efficient as their specialized counterparts, they can certainly provide the flexibility needed in practice.

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 steering the test process to avoid consequences of unanticipated events, so that testing objectives can be achieved in a cost-effective manner, are key aspects of practical CIT.

Test results, of course, need to be analyzed, and here we have focused only on fault-characterization. However, one interesting avenue for future research would be 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) and yet improve accuracy (compared to probabilistic approaches). In general, better tool support for detecting and locating faulty interactions, as well as for assessing the thoroughness of interaction testing, are of practical importance. C

Acknowledgments

This research was supported by a Marie Curie International Reintegration Grant within the Seventh European Community Framework Programme (FP7-PEOPLE-IRG-2008), by the Scientific and Technological Research Council of Turkey (109E182), and by US NSF awards (CCF-0811284, CCF-0747009, CCF-1161767, and CCF-1116740) and an AFOSR award (FA9550-10-1-0406).

References

1. D.R. Kuhn et al., "Combinatorial Methods for Event Sequence Testing," *Proc. 5th IEEE Int'l Conf. Software Testing, Verification and Validation* (ICST 12), 2012, pp. 601–609.
2. C. Nie and H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys*, vol. 43, no. 11, 2011, pp. 1–29.
3. C. Martinez et al., "Locating Errors Using ELAs, Covering Arrays, and Adaptive Testing Algorithms," *SIAM J. Discrete Mathematics*, vol. 23, no. 4, 2009, pp. 1776–1799.
4. C. Yilmaz, "Test Case-Aware Combinatorial Interaction Testing," *IEEE Trans. Software Eng.*, vol. 39, no. 5, 2013, pp. 684–706.
5. H. Srikanth, M.B. Cohen, and X. Qu, "Reducing Field Failures in System Configurable Software: Cost-Based Prioritization," *Proc. 20th IEEE Int'l Symp. Software Reliability Eng.* (ISSRE 09), 2009, pp. 61–70.
6. G. Demiroz and C. Yilmaz, "Cost-Aware Combinatorial Interaction Testing," *Proc. 2012 Int'l Conf. Advances in*

System Testing and Validation Lifecycle (VALID 12), 2012, pp. 9–16.
7. X. Yuan, M.B. Cohen, and A.M. Memon, "GUI Interaction Testing: Incorporating Event Context," *IEEE Trans. Software Eng.*, vol. 37, no. 4, 2011, pp. 559–574.
8. S. Fouché, M.B. Cohen, and A. Porter, "Incremental Covering Array Failure Characterization in Large Configuration Spaces," *Proc. 18th Int'l. Symp. Software Testing and Analysis* (ISSTA 09), 2009, pp. 177–188.
9. A. Calvagna and E. Tramontana, "Incrementally Applicable T-Wise Combinatorial Test Suites for High-Strength Interaction Testing," *Proc. 37th IEEE Computer Software and Applications Conf. Workshops* (COMPSAC 13), 2013, pp. 77–82.
10. C. Song, A. Porter, and J.S. Foster. "Efficiently Discovering High-Coverage Configurations Using Interaction Trees," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 903–913.
11. R.C. Bryce and C.J. Colbourn, "Prioritized Interaction Testing for Pair-Wise Coverage with Seeding and Constraints," *Information and Software Technology*, vol. 48, no. 10, 2006, pp. 960–970.
12. E. Dumlu et al., "Feedback-Driven Adaptive Combinatorial Testing," *Proc. 20th Int'l Symp. Software Testing and Analysis* (ISSTA 11), 2011, pp. 243–253.

Cemal Yilmaz *is an assistant professor of computer science at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. His current research focuses on software testing; dynamic analysis for fault detection, prediction, and localization; and developing approaches and tools to improve software quality. Yilmaz received a PhD in computer science from the University of Maryland, College Park. He is a member of IEEE. Contact him at cyilmaz@sabanciuniv.edu.*

Sandro Fouché *is an assistant professor in the Department of Computer and Information Sciences at Towson University. His current research focuses on software tools for dynamic fault analysis, distributed systems, and reverse engineering. Fouché received a PhD in computer science from the University of Maryland, College Park. He is a member of IEEE and ACM. Contact him at sfouche@towson.edu.*

Myra Cohen *is an associate professor in the Department of Computer Science and Engineering at the University of Nebraska–Lincoln. Her current research includes testing of highly configurable software systems, combinatorial interaction testing, and search-based software engineering. Cohen received a PhD in computer science from the University of Auckland, New Zealand. She is a member of IEEE and the ACM. Contact her at myra@cse.unl.edu.*

Adam A. Porter *is a professor of computer science at the University of Maryland and the University of Maryland Institute for Advanced Studies (UMIACS). His research interests include developing tools and techniques for large-scale software development. Porter received a PhD in computer science from the University of California at Irvine. He is a senior member of IEEE and ACM. Contact him at aporter@cs.umd.edu.*

Gulsen Demiroz *is an instructor and PhD student in computer science at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. Her current research focuses on software testing, software quality assurance, and metaheuristic approaches for computing covering arrays. Demiroz received an MS in computer science from Bilkent University, Ankara, Turkey. Contact her at atgulsend@sabanciuniv.edu.*

Ugur Koc *is a graduate student at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. His current research focuses on software quality assurance. Koc received a BS in computer science and engineering from Fatih University, Istanbul, Turkey. Contact him at ugurkoc@sabanciuniv.edu.*