A Constraint Solving Problem Towards Unified Combinatorial Interaction Testing

Hanefi Mercan and Cemal Yilmaz

Faculty of Engineering and Natural Sciences,
Sabanci University, Istanbul, Turkey
{hanefimercan,cyilmaz}@sabanciuniv.edu

Abstract

Combinatorial Interaction Testing (CIT) approaches aim to reveal failures caused by the interactions of factors, such as input parameters and configuration options. Our ultimate goal in this line of research is to improve the practicality of CIT approaches. To this end, we have been working on developing what we call *Unified Combinatorial Interaction Testing* (U-CIT), which not only represents most (if not all) combinatorial objects that have been developed so far, but also allows testers to develop their own application-specific combinatorial objects for testing. However, realizing U-CIT in practice requires us to solve an interesting constraint solving problem. In this work we informally define the problem and present a greedy algorithm to solve it. Our gaol is not so much to present a solution, but to introduce the problem, the solution of which (we believe) is of great practical importance.

1 Introduction

Software systems frequently embody a wide spectrum of system variabilities that require testing, such as software and hardware configuration options, user inputs, and thread interleavings. However, exhaustively testing all possible variations in a timely manner (if not impossible at all) is generally far beyond the available resources for testing [15]. For this reason, the testing of modern software systems almost always involve sampling enormous variability 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 (or CIT) [15, 10].

CIT approaches work by first defining a model of the system's variability space. This model typically includes a set of factors, each of which takes its value from a particular domain, and a (possibly empty) set of inter-option constraints, each of which invalidates certain factor value combinations, as not all possible combinations may be valid in practice. Based on this model, CIT then generates a sample, meeting a specified *coverage criterion.* That is, the sample contains some specified combinations of factors and their values. For instance, a *t-way covering array*, which is a well-known and frequently-used CIT object, requires that each valid combination of factor values for every combination of t factors, appears at least once in the sample [2]. Here, t is often referred to as the coverage strength.

The basic justification for using CIT is that they can (under certain assumptions) effectively and efficiently exercise all system behaviors caused by the interactions of t or fewer factors. The effectiveness of CIT stems from the coverage properties it provides; e.g., all required t-way combinations of factor values are guaranteed to be covered at least once. The efficiency, on the other hand, stems from the fact that a test case can cover more than one required combination. For example, a configuration composed of n configuration options covers $\binom{n}{t}$ different t-way option setting combinations. That is, testing a single configuration has the potential of testing all $\binom{n}{t}$ combinations. Therefore, carefully generating test cases, such that a full coverage under the given coverage criterion is obtained using a minimum number of test cases can, for example, decrease the cost of testing.

The results of many empirical studies suggest that majority of factor-related failures in practice are caused by the interactions of only a small number of factors. That is, t is small in practice, typically $2 \leq t \leq 6$ with t=2 (i.e., pairwise testing) being the most common case [3, 2, 4, 5]. For a fixed t, as the variability space grows (as the number of factors increases, for example), the size of a CIT object represents an increasingly smaller proportion of the whole space. Thus, very large spaces can efficiently be covered. Consequently, CIT has been successfully used in many application domains, including systematic testing of network protocols [1, 12], input parameters [11], configurations [8, 14], software product lines [7], multi-threaded applications [9], and graphical user interfaces [16].

All these have so far been achieved by having researchers develop specific models for defining variability spaces together with specific coverage criteria to be used for testing, which in turn led to the development of novel CIT objects for every unique testing scenario. However, when the testing scenarios encountered in practice deviate from the ones addressed by researchers, practitioners often have profound difficulties in using these existing CIT objects [15]. As a matter of fact, even small changes in variability spaces and/or coverage criteria, may render existing CIT objects useless. For example, the very first variants of covering arrays supported only pairwise testing (where t=2) of binary factors. That is, when t>2 and/or when factors had varying number of values, these CIT objects were useless. Then, new CIT objects were developed to handle scenarios, in which factors might have different number of values and the covering arrays could be created for $t \geq 2$. But then these new objects suffered in the presence of inter-option constraints. New combinatorial objects were developed to handle system-wide constraints, but they then suffered in the presence of test case-specific inter-option constraints, which led to the development of test case-aware covering arrays [13]. This trend has been going on for decades now. And the bad news is that every time the variability space and/or the coverage criterion changes, a new CIT object needs to be defined, which in turn necessitates the development of specialized construction approaches, algorithms, and tools to compute these objects. Clearly, all these have been greatly hindering the applicability of CIT in practice.

We conjecture that the flexibility, thus the applicability of CIT in practice, would greatly be improved, if there were better tools that allowed practitioners to define their own application-specific variability spaces as well as their own application-specific coverage criteria. That is, rather than we, as researchers, invent new CIT objects for testing and ask practitioners to use them, thus telling them what to test, we would like to enable practitioners to define their own space for testing as well as their own coverage criterion, thus enabling them to invent their own application-specific CIT objects. Our goal as researchers is then to develop powerful tools to efficiently and effectively sample the given space to obtain full coverage under the given criterion. Although such generic tools may not be as efficient as their specialized counterparts, they certainly can provide the flexibility needed in practice. We call this approach *Unified Combinatorial Interaction Testing* (or U-CIT).

In this work we informally introduce U-CIT, which enables practitioners to define their own variability spaces and coverage criteria for testing, and present a unified construction approach to compute specific instances of U-CIT objects.

2 Unified Combinatorial Interaction Testing (U-CIT)

Definition 1. *A U-CIT requirement is an entity that needs be covered. In U-CIT, requirements are expressed as constraints.*

For example, for scenarios, in which standard covering arrays are used for testing highly configurable systems, a U-CIT requirement corresponds to a t-tuple to be covered, where a t-tuple is a set of option-setting pairs for a combination of t distinct configuration options.

Definition 2. *A U-CIT test case is a collection of U-CIT test requirements that can be tested together, i.e., a set of constraints that can be satisfied together.*

In our running scenario, for example, a U-CIT test case corresponds to a system configuration, which is indeed an n-tuple, where n is the number of configuration options.

Definition 3. *A U-CIT space model is a system of constraints that implicitly define the space of all valid U-CIT requirements as well as all valid U-CIT test cases, as not all possible combinations of U-CIT requirements may be valid in practice.*

For our running scenario, a U-CIT space model specifies that 1) every configuration option must have a valid setting in a configuration, 2) a valid U-CIT requirement is a valid t-tuple that does not violate any inter-option constraints, and 3) a valid U-CIT test case is valid configuration that does not violate any inter-option constraints.

Definition 4. *A U-CIT coverage criterion is a criterion that implicitly defines all valid U-CIT requirements that need to be covered.*

For example, the U-CIT coverage criterion for our running scenario states that all valid t-tuples must be covered at least once.

U-CIT takes as input a U-CIT space model and a U-CIT coverage criterion and as output computes a U-CIT object, e.g., a set of valid U-CIT test cases, which achieves a full coverage under the given criterion. Although it is possible to define additional constraints on the emergent properties of the resulting objects, such as the objects must achieve a full coverage with the "minimum" possible testing cost [6], we, for this work, assume one such emergent constraint which aims to minimize the number of test cases required for full coverage.

What makes a U-CIT approach a unified approach is that requirements to be covered, test cases, and the space from which the test cases will be drawn, are all expressed as constraints. Consequently, the problem of computing a U-CIT object turns into one big, interesting constraint solving problem. Note that we use the term "constraint" in the general sense; any restriction, independent of the logic in which it is specified, is considered to be a constraint. In other words, no matter whether the constraints are specified using Boolean logic, first-order logic, temporal logic, etc., the proposed approach will work as long as an appropriate constraint solver is provided.

Algorithm 1 An algorithm for computing U-CIT objects

```
Input: A U-CIT space model $M$, a U-CIT coverage criterion $C$
Output: A U-CIT object $O$
 1: ▷ Determine all valid U-CIT requirements
 2: $R \leftarrow \{\}$
 3: for each requirement $r$ implied by $C$ do
 4:     if $isSatisfiable(r \wedge M)$ then
 5:         $R \leftarrow R \cup r$
 6:     end if
 7: end for
 8:
 9: ▷ Compute a "minimum" number of satisfiable subsets
10: $S \leftarrow \{\}$
11: for each $r \in R$ do
12:     $accommodated \leftarrow false$
13:     for each $R' \in S$ do
14:         if $isSatisfiable(r \wedge M \wedge \bigwedge_{r' \in R'} r')$ then
15:             $R' \leftarrow R' \cup \{r\}$
16:             $accommodated \leftarrow true$
17:             break
18:         end if
19:     end for
20:     if not $accommodated$ then
21:         $S \leftarrow S \cup \{\{r\}\}$
22:     end if
23: end for
24:
25: ▷ Generate the actual test cases
26: $O \leftarrow \{\}$
27: for each $R' \in S$ do
28:     $\tau \leftarrow solve(M \wedge \bigwedge_{r' \in R'} r')$
29:     $O \leftarrow O \cup \tau$
30: end for
31: return $O$
```

3 Constraint Satisfaction Problem

A U-CIT coverage criterion effectively defines a set of constraints to be satisfied (not necessarily all together, but in groups), each of which represents a U-CIT requirement. Given the requirements to be covered and a U-CIT space model further constraining the variability space from which the U-CIT test cases will be drawn, the constraint satisfaction problem we need to solve is to divide the requirements into a minimum number of non-overlapping sets of requirements, such that within each set, the constraints representing the requirements in the set as well as the model constraints are satisfiable together. In effect, a solution for each set represents a valid U-CIT test case, i.e., a collection of U-CIT requirements that can be tested together. Therefore, the test cases generated for all the sets, represent a U-CIT object achieving full coverage under the given coverage criterion. In particular, by reducing the number of non-overlapping sets, thus the number of test cases, we attempt to reduce testing costs.

4 A Greedy Approach for Computing U-CIT Objects

In this section we present a greedy algorithm (Algorithm 1) to compute U-CIT objects. Given a U-CIT space model M and a coverage criterion C, we first determine all valid U-CIT requirements R (lines 2-7). To this end, we enumerate all the entities to be covered, convert each entity to a constraint r, and then determine whether $r \wedge M$ is satisfiable (line 4). If it is, then r is added in R (line 5). Otherwise, r is invalid.

Once the set of valid requirements R is determined, we divide it into non-overlapping satisfiable subsets S, covering all requirements (lines 9-23). To this end, we start with an empty pool of subsets (line 10). Then, for each requirement r in R, we attempt to accommodate it in an existing subset in the pool (line 14). If such a subset is found, we include r in the subset (line 15). If not, we populate the pool with an initially empty subset and then include r in the newly added subset (line 21). Note that a subset of requirements R' in this context is specified as the logical conjunction of all the requirements included in the set, i.e., $\bigwedge_{r' \in R'} r'$. Consequently, to determine whether a new requirement r can be accommodated in an existing subset R', we solve the respective constraints together with M, i.e., $r \wedge M \wedge \bigwedge_{r' \in R'} r'$ (line 14), if the resulting constraint is satisfiable then we include r in R' (line 15).

After determining the subsets S, to compute the U-CIT object O, we generate a test case by solving the logical conjunction $M \wedge \bigwedge_{r \in R'} r$ for each subset R' (line 28). The set of test cases are then guaranteed to obtain full coverage under the coverage criterion C.

Not that we provide this algorithm as a proof-of-concept algorithm for computing U-CIT objects. That is, in the development of this algorithm, our major concern was correctness, not performance. Consequently, the proposed algorithm suffers from some drawbacks. One issue is that being a greedy algorithm, it may yield locally optimal solutions. Another issue is that the same constraints may end up being solved repeatedly, which may cause scalability issues.

5 An Example: Specifying and Computing Standard Covering Arrays as U-CIT Objects

In this section, we illustrate U-CIT on a hypothetical system by providing details about how our running scenario, in which standard covering arrays are used for configuration testing, can be handled by U-CIT. The example is kept as simple as possible on purpose. In general, the complexity of encodings depends on the complexity of the system under test and/or complexity of the application domain.

Without losing the generality of the proposed approach, the system under test we use in our example has three binary configuration options (o_1, o_2, and o_3), each of which takes the setting of *true* or $false$, together with two inter-option constraints: $o_2 = true \rightarrow o_3 = true$, invalidating the combination ($o_2 = true$, $o_3 = false$), and $\neg(o_1 = true \wedge o_3 = false)$, invalidating the combination ($o_1 = true$, $o_3 = false$). Furthermore, the system is to be tested using a 2-way covering array, i.e., all valid 2-tuples must be covered at least once.

For this example, the U-CIT space model M can be expressed in Boolean algebra as $(\neg o_2 \vee o_3) \wedge (\neg o_1 \vee o_3)$, where each configuration option is represented by a Boolean variable. In this encoding, each U-CIT requirement simply becomes a Boolean formula representing a 2-tuple. For example, the 2-tuple ($o_1 = false$, $o_2 = true$) is expressed as $(\neg o_1 \wedge o_2)$. To determine whether a U-CIT requirement is valid or not, it is checked whether the respective Boolean formula is satisfiable with the U-CIT space model M. For example, since $(\neg o_1 \wedge o_2) \wedge M$ is

(o_1, o_2)	(o_1, o_3)	(o_2, o_3)
$r_1 : \neg o_1 \wedge \neg o_2$	$r_5 : \neg o_1 \wedge \neg o_3$	$r_8 : \neg o_2 \wedge \neg o_3$
$r_2 : \neg o_1 \wedge o_2$	$r_6 : \neg o_1 \wedge o_3$	$r_9 : \neg o_2 \wedge o_3$
$r_3 : o_1 \wedge \neg o_2$		
$r_4 : o_1 \wedge o_2$	$r_7 : o_1 \wedge o_3$	$r_{10} : o_2 \wedge o_3$

Table 1: All valid U-CIT requirements.

satisfiable, the 2-tuple ($o_1 = false$, $o_2 = true$) is a valid 2-tuple, thus a valid U-CIT requirement. For the same reason, ($o_2 = true$, $o_3 = false$) is not a valid U-CIT requirement.

The first part of Algorithm 1 (lines 2-7), which determines all valid U-CIT requirements, would then generate the 10 U-CIT requirements $r_1, \cdots, r_{10}$ given in Table 1.

	2-way covering array		
$S = \{R'_1, R'_2, R'_3, R'_4\}$	o_1	o_2	o_3
$R'_1 = \{r_1, r_5, r_8\}$	false	false	fase
$R'_2 = \{r_2, r_6\}$	false	true	true
$R'_3 = \{r_3, r_7, r_9\}$	true	false	true
$R'_4 = \{r_4, r_{10}\}$	true	true	true

Table 2: The set of requirements divided into non-overlapping satisfiable subsets of requirements $S = \{R'_1, R'_2, R'_3, R'_4\}$ (left column), and the respective standard 2-way covering array generated (right column).

Next, the set of valid requirements is divided into non-overlapping satisfiable subsets by the second part of Algorithm 2 (lines 10-23). Assuming that the requirements in Table 1 are processed in the order $r_1, \cdots, r_{10}$, the first requirement we process becomes $r_1 : (\neg o_1 \wedge \neg o_2)$. Since the set S is initially empty, we create a new subset $R'_1 = \{r_1\}$ and populate S with R'_1, i.e., $S = \{R'_1\}$. For the second requirement $r_2 : (\neg o_1 \wedge o_2)$, since $r_1 \wedge r_2 \wedge M$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge o_2) \wedge M$, is not satisfiable, r_2 cannot be placed in R'_1. So, we create a new subset $R'_2 = \{r_2\}$ and S becomes $\{R'_1, R'_2\}$. After processing r_4, we would have four subsets R'_1, R'_2, R'_3, and R'_4 in S, containing requirements r_1, r_2, r_3, and r_4, respectively. For requirement $r_5 : (\neg o_1 \wedge \neg o_3)$, as $r_1 \wedge r_5 \wedge M$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge \neg o_3) \wedge M$, is satisfiable r_5 is placed in R'_1 together with r_1. After processing all the requirements, we would have the four subsets of satisfiable U-CIT requirements given in the left column of Table 2.

Finally, in the last part of Algorithm 1 (lines 26-30), for each subset of U-CIT requirements in S, we generate a U-CIT test case, which in this case corresponds to a valid configuration, by solving the requirements in the subset together with the U-CIT space model M. For example, for R'_1 (Table 2), solving $r_1 \wedge r_5 \wedge r_8 \wedge M$ produces the configuration $(o_1 = false, o_2 = false, o_3 = false)$. Solving all the subsets would then generate the U-CIT object given in the right column of Table 2, which is indeed a standard 2-way covering array – a set of configurations that covers each valid 2-tuple at least once.

6 Conclusion and Future Work

We believe that U-CIT can greatly improve the flexibility of combinatorial interaction testing in practice. Therefore, efficient and affective approaches for solving the constraint satisfaction problem we informally introduced in this paper, are of great practical importance. Therefore, we keep on developing languages and model-based tools for defining variability spaces and coverage criteria in a generic manner as well as developing efficient and effective approaches for computing U-CIT objects.

References

[1] Kirk Burroughs, Aridaman Jain, and Robert L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Communications, 1994. ICC'94, SUPERCOMM/ICC'94, Conference Record,'Serving Humanity Through Communications.'IEEE International Conference on*, pages 745–752. IEEE, 1994.

[2] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997.

[3] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE software*, 13(5):83–88, 1996.

[4] Jacek Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430, 2006.

[5] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, and Christopher M. Lott. Model-based testing of a highly programmable system. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 174–179. IEEE, 1998.

[6] Gulsen Demiroz and Cemal Yilmaz. Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles*, pages 9–16, 2012.

[7] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.

[8] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[9] Yu Lei, Richard H Carver, Raghu Kacker, and David Kung. A combinatorial testing strategy for concurrent programs. *Software Testing, Verification and Reliability*, 17(4):207–225, 2007.

[10] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[11] Patrick J. Schroeder, Pat Faherty, and Bogdan Korel. Generating expected results for automated black-box testing. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 139–148. IEEE, 2002.

[12] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 246–254. IEEE, 1996.

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

[14] Cemal Yilmaz, Myra B. Cohen, Adam Porter, et al. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[15] Cemal Yilmaz, Sandro Fouche, Myra B. Cohen, Adam Porter, Gulsen Demiroz, and Ugur Koc. Moving forward with combinatorial interaction testing. *Computer*, 47(2):37–45, 2014.

[16] Xun Yuan, Myra B. Cohen, and Atif M. Memon. Gui interaction testing: Incorporating event context. *Software Engineering, IEEE Transactions on*, 37(4):559–574, 2011.