UNIFIED COMBINATORIAL INTERACTION TESTING

by
HANEFI MERCAN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Doctor of Philosophy

Sabancı University
June 2021

UNIFIED COMBINATORIAL INTERACTION TESTING

Approved by:

Date of Approval: June 22, 2021

ABSTRACT

UNIFIED COMBINATORIAL INTERACTION TESTING

HANEFI MERCAN

Computer Science and Engineering Ph.D Dissertation, June 2021

Dissertation Supervisor: Assoc. Prof. Cemal Yılmaz

Keywords: Combinatorial interaction testing, covering arrays, sequence covering arrays, constraint solving, structural coverage, coverage criteria

We present Unified Combinatorial Interaction Testing (U-CIT), which aims to improve the flexibility of combinatorial interaction testing (CIT) by eliminating the necessity of developing specialized constructors for CIT problems that cannot be efficiently and effectively addressed by the existing CIT constructors. U-CIT expresses the entities to be covered and the space of valid test cases, from which the samples are drawn to obtain full coverage, as constraints. Computing a U-CIT object (i.e., a set of test cases obtaining full coverage under a given coverage criterion) then turns into an interesting constraint solving problem, which we call *cov-CSP*. cov-CSP aims to divide the constraints, each representing an entity to be covered, into a minimum number of satisfiable clusters, such that a solution for a cluster represents a test case and the collection of all the test cases generated (one per cluster) constitutes a U-CIT object, covering each required entity at least once. To solve the cov-CSP problem, thus to compute U-CIT objects, we first present two constructors. One of these constructors attempts to cover as many entities as possible in a cluster before generating a test case, whereas the other constructor generates a test case first and then marks all the entities accommodated by this test case as covered. We then use these constructors to evaluate U-CIT in three studies, each of which addresses a different CIT problem. In the first study, we develop structure-based U-CIT objects to obtain decision coverage-adequate test suites. In the second study, we develop order-based U-CIT objects, which enhance a number of existing order-based coverage criteria by taking the reachability constraints imposed by graph-based models directly into account when computing interaction test suites. In the third study, we develop usage-based U-CIT objects to address the scenarios, in which standard

covering arrays are not desirable due to their sizes, by choosing the entities to be covered based on their usage statistics collected from the field. Then, we empirically demonstrated that the performance (i.e., the construction times) of U-CIT constructors can be significantly improved by using *hints*. We also carry out user studies to further evaluate U-CIT. The results of these studies suggest that U-CIT is more flexible than the existing CIT approaches.

ÖZET

TÜMLEŞİK KOMBİNEZON ETKİLEŞİM SINAMA YÖNTEMİ

HANEFI MERCAN

Bilgisayar Bilimi ve Muhendisliği Doktora Tezi, Haziran 2021

Tez Danışmanı: Doç Dr. Cemal Yılmaz

Anahtar Kelimeler: Kombinezon etkileşim sınaması, kapsayan diziler, sıralı kapsayan diziler, kısıt çözümü, yapısal kapsama, kapsama kriteri

Önermiş olduğumuz Tümleşik Kombinezon Etkileşim Sınama (T-KES) yöntemi, mevcut Kombinezon Etkileşim Sınama (KES) yöntemleri ile verimli ve etkili bir şekilde çözülemeyen KES problemleri için özel hesaplama yöntemleri geliştirme gerekliliğini ortadan kaldırarak, KES'in esnekliğini arttırmayı amaçlamaktadır. T-KES kapsanması gereken test edilebilen isterleri ve test havuzunun (yani T-KES objesinin) oluşturulacağı test durumları uzayını kısıt olarak ifade etmektedir. Böylece bir T-KES objesi oluşturma problemi (yani, belirli bir kapsama kriteri altında tam kapsama elde eden test durumları kümesi) bizim *cov-CSP* olarak adlandırdığımız ilginç bir kısıt çözme problemine dönüşmektedir. cov-CSP kapsanması gereken her isteri temsil eden kısıtları minimum sayıda kümelere bölmeyi hedeflemektedir. Öyle ki, bu kümelerin her birisi daha sonra bir test durumunu ifade edecek olup, üretilen tüm bu test durumlarından (her kısıt kümesi için bir tane) oluşan test havuzu ise T-KES objesini oluşturmaktadır. Böylece T-KES objesi her isterin en azından bir test durumu tarafından kapsandığını garanti etmektedir. Tez kapsamında, cov-csp problemini çözmek ve dolayısıyla T-KES objelerini üretebilmek için iki tane hesaplama yöntemi önerilmektedir. Bu hesaplama yöntemlerinden biri, bir test durumu oluşturulmadan önce bir kümede mümkün olduğunca çok fazla isteri kapsamaya çalışırken, diğer yöntem ise önce bir test durumu oluşturur ve ardından bu test durumunun kapsadığı tüm isterleri kapsanmış olarak işaretlemektedir. Akabinde, bu hesaplama yöntemleri kullanılarak her biri farklı KES problemini çözmeye çalışan 3 farklı çalışmada T-KES yaklaşımı test edilmiştir. İlk çalışmada, karar kapsaması tabanlı yapısal T-KES objeleri üretilmiştir. İkinci çalışmada, çizge tabanlı modellerin getirdiği erişilebilirlik kısıtlarını dikkate alarak bir takım sıralama

tabanlı kapsama kriteri geliştirilmiş ve bu kapsama kriterleri ile sıralama tabanlı T-KES objeleri üretilmiştir. Üçüncü çalışmada ise, standart kapsayan dizilerin çok sayıda test durumları üretmesinden ötürü kullanılamadıklarından, sahadan toplanan kullanım istatistiklerine göre kapsanması gereken isterler seçilerek kullanıma dayalı T-KES objeleri geliştirilmiştir. Sonrasında, yeni önerdiğimiz ipucu kavramının T-KES'in etkinliğini daha da arttırdığını göstermek adına bir takım deneysel çalışmalar yapılmıştır. Son olarak T-KES'i daha ileri düzeyde değerlendirebilmek için saha çalışmaları da yapılmıştır. Bu çalışmaların sonuçları, T-KES'in mevcut KES yaklaşımlarından daha esnek olduğunu göstermektedir.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Prof. Cemal Yilmaz whose expertise, understanding, and patience, added considerably to my graduate experience. During my Ph.D. education, he gave me the moral support and the freedom I needed to move on. It has been a great honor to work under his guidance.

I also would like to extend my sincere thanks to the members of my dissertation committee, Prof. Esra Erdem Patoğlu, Prof. Bülent Çatay, Prof. Myra B. Cohen, and Prof. Alper Şen for serving as my committee members, their contributions and insightful comments. I also gratefully acknowledge the assistance of Prof. Hüsnü Yenigün and Prof. Kamer Kaya.

A special thanks go to Kağan Aksoydan for supporting me throughout the years with his friendship and for all the fun we had.

Moreover, this research was supported by the Scientific and Technological Research Council of Turkey TUBITAK (118E204). I would also like to acknowledge TUBITAK for supporting me financially during my Ph.D. education with a scholarship program (BIDEB 2211).

Most importantly, none of this would have been possible without the love and patience of my parents Hatice Mercan and Ahmet Remzi Mercan for supporting spiritually me throughout my life. Lastly, my deepest and heartiest thanks to my dear wife Sevinç Mercan and my little handsome kid Enes Ali Mercan for being in my life and empowering me with their great love.

To my lovely family

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

1. INTRODUCTION

Exhaustively testing the input spaces of modern software systems in a timely manner (if not impossible at all) is generally far beyond the available resources for testing (Yilmaz, Fouche, Cohen, Porter, Demiroz & Koc, 2014), such as time, computers, storage devices, network resources, and person-hour. Combinatorial interaction testing (CIT) approaches systematically sample the input space and test only the selected instances of the system's behavior (Nie & Leung, 2011; Yilmaz et al., 2014). Note that the term "input" in CIT is used in the most general sense to refer to any factor, which can affect program executions, such as configuration options, input parameters, user events, etc.

CIT approaches typically model the software under test as a set of parameters, each of which takes its values from a discrete domain. As not all possible combinations of parameter values may be valid in practice, the model can also have a set of constraints, which invalidate certain combinations. Based on this model, CIT then generates a sample, i.e., a set of test cases, which from now on will be referred to as a *CIT object*, meeting a specified *coverage criterion*. That is, the sample contains some specified combinations of parameters and their values. For instance, *t-way covering arrays* – a well-known CIT approach, where t is called the *coverage strength* – requires that each valid combination of parameter values for every combination of t parameters appears at least *once* in the sample (Cohen, Dalal, Fredman & Patton, 1997), aiming to reveal all the failures caused by the interactions of t or fewer parameters.

As an example, which will further be studied in detail in Chapter 4.1, Figure 1.1a presents a configurable system with 6 compile-time configuration options $(o_1, \ldots, o_6)$ implemented by using preprocessor directives. Each option has two levels of settings {*(T)rue*, *(F)alse*} and there are no inter-option constraints (i.e., all combinations of option settings are valid). The set of test cases in Figure 1.1b represent a 2-way covering array, i.e., a CIT object, for this system. Since $t = 2$, all pairwise combinations of settings for these 6 configuration options can be found in at least one of the 7 test cases selected by this CIT object.

```
1  #ifdef (o1 && o2)
2    #ifdef (o3 || o4)
3      ...
4    #endif
5  #endif

6  #ifdef (o5)
7    #ifdef (o6)
8      ...
9    #endif
10 #endif
```

(a)

test cases						decision outcomes			
o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
T	F	T	F	F	F	*F*	-	F	-
F	T	F	T	T	F	*F*	-	T	F
T	T	T	T	F	T	*T*	*T*	F	-
T	F	F	F	T	F	*F*	-	T	F
F	F	T	F	T	T	*F*	-	T	T
F	F	F	T	F	T	*F*	-	F	-
T	T	T	F	F	T	*T*	*T*	F	-

(b)

entities to be covered
$e_1 : (o_1 \wedge o_2)$
$e_2 : \neg(o_1 \wedge o_2)$
$e_3 : (o_1 \wedge o_2) \wedge (o_3 \vee o_4)$
$e_4 : (o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)$
$e_5 : (o_5)$
$e_6 : (\neg o_5)$
$e_7 : (o_5 \wedge o_6)$
$e_8 : (o_5 \wedge \neg o_6)$

(c)

test cases						decision outcomes			
o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
T	T	T	T	T	T	*T*	*T*	*T*	*T*
F	F	T	F	F	T	*F*	-	*F*	*T*
T	T	F	F	T	F	*T*	*F*	*T*	*F*

(d)

Figure 1.1 (a) An example set of preprocessor directives for a system with 6 compile-time configuration options, (b) an example 2-way standard covering array created for the system, (c) entities to be covered to obtain full coverage under the decision coverage criterion, and (d) an example test suite obtaining full coverage under the decision coverage criterion.

To reduce the cost of testing, CIT constructors, i.e., the tools to compute CIT objects, aim to obtain a full coverage under the given criterion by using the smallest number of test cases possible. CIT has indeed been successfully used in many application domains, including systematic testing of network protocols (Williams & Probert, 1996), input parameters (Schroeder, Faherty & Korel, 2002), software configurations (Yilmaz, Cohen & Porter, 2006a), software product lines (Johansen, Haugen & Fleurey, 2012), multi-threaded applications (Lei, Carver, Kacker & Kung, 2007), and graphical user interfaces (Yuan et al., 2011).

We, however, observe that when the actual CIT problems differ from the ones addressed by the existing CIT approaches, it can be difficult to use these approaches in an efficient and effective manner (Demiroz & Yilmaz, 2012; Yilmaz, 2013a; Yilmaz et al., 2014). Note that, in this context, changes in CIT problems refer to changes in the coverage criteria or in the properties of the test spaces from which the samples are drawn, such that existing CIT constructors cannot be used as they are (i.e., requiring modifications, if at all possible) or demand excessive number of test cases

to guarantee full coverage.

For example, if the coverage criterion in our running example was changed from t-way coverage to decision coverage (Javeed & Yilmaz, 2015), where the goal is to cover every outcome of a decision in Figure 1.1a at least once, then, to guarantee full coverage, the strength of the standard covering array to be used would be at least 4 (i.e., $t \geq 4$). This is because the outcome of the decision in line 2 (Figure 1.1a) depends on the interactions among 4 options, namely o_1, o_2, o_3, and o_4. This, however, requires to have at least 16 test cases, while a full decision coverage in this scenario can be achieved by using as little as 3 test cases, such as the ones given in Figure 1.1d.

Different CIT problems typically necessitate the development of specialized constructors. Taking a brief look at the historical perspective of covering arrays can help understand this trend: The very first variants of covering array constructors supported only pairwise testing of binary parameters, where $t = 2$ and each parameter had exactly two levels of values (Lawrence, Kacker, Lei, Kuhn & Forbes, 2011). When these strict conditions were not met, the aforementioned objects were of little worth. Consequently, new CIT constructors were developed to handle the CIT problems, in which the parameters could take on a different number of values and the covering arrays could be computed for $t \geq 2$ (Cohen et al., 1997). However, as these objects assumed that all possible combinations of parameter values were valid, they were not appropriate in the presence of system-wide inter-parameter constraints, causing wasted resources in testing (Cohen, Dwyer & Shi, 2007; Dumlu, Yilmaz, Cohen & Porter, 2011). Thus, new CIT constructors were developed to handle system-wide constraints (Bryce & Colbourn, 2006; Mats, Jeff & Jonas, 2006). However, these objects then became inappropriate in the presence of test case-specific constraints, which led to the development of test case-aware covering arrays and their specialized constructors (Yilmaz, 2013b).

Developing specialized constructors can, however, be quite challenging and time-consuming, which is also apparent from more than 50 papers published in the literature, the sole purpose of which is to compute standard covering arrays (Nie & Leung, 2011; Yilmaz et al., 2014).

In this thesis, we introduce *Unified Combinatorial Interaction Testing* (U-CIT) to improve the flexibility of CIT by eliminating the necessity of developing specialized constructors for every distinct CIT problem. In U-CIT, both the entities to be covered and the space of test cases, from which the samples will be drawn, are expressed as constraints. The problem of computing an U-CIT object to cover all the requested entities then turns into an interesting constraint solving problem, which

we call *cov-CSP* (Makaś, 2016; Rescher & Manor, 1970; Schotch & Jennings, 1980). Given a set of constraints, each of which represents an entity to be covered, cov-CSP aims to divide the constraints into a minimum number of satisfiable clusters, such that each cluster depicts a subset of the entities, which can be tested together in a single test case. A solution for a cluster then represents a test case, covering all the entities included in the cluster. Consequently, the collection of all the test cases generated (one per cluster) constitutes an U-CIT object that covers each required entity at least once. In the remainder of the thesis, we use the terms "CIT object" and "U-CIT object" interchangeably to refer to a set of test cases, which obtain full coverage under a given coverage criterion.

Going back to our running example (Figure 1.1), a decision coverage-adequate U-CIT object can be computed by representing each configuration option as a Boolean variable. Then, each entity to be covered corresponds to a distinct outcome of a decision, represented as a constraint in Boolean logic. Figure 1.1c presents all the entities that need to be covered to obtain full decision coverage for the system given in Figure 1.1a. These entities can be divided into 3 satisfiable clusters: $\{e_1, e_3, e_5, e_7\}$, $\{e_2, e_6\}$, and $\{e_4, e_8\}$. A solution for each cluster represents a test case. For example, the three test cases in Figure 1.1d, each of which corresponds to a solution computed for a distinct cluster, represent an U-CIT object, achieving full decision coverage.

Note that we use the term "constraint" in the general sense in U-CIT. That is, any restriction, independent of the logic in which it is specified, is considered to be a constraint. Consequently, an U-CIT constructor can be used as long as the entities to be covered are expressed as constraints and an appropriate procedure (i.e., a "solver") is provided to determine if a given set of entities can be tested together in a single test case, i.e., if the respective constraints can be satisfied together. In our running example (Figure 1.1), for instance, we can use an ordinary SAT or CSP solver (Biere, Heule & van Maaren, 2009) to figure out whether the constraints included in the clusters are satisfiable or not. We, therefore, believe that U-CIT can be used in a wide spectrum of domains, including software product lines, system of systems, and cyber-physical systems, in addition to the domains, which we used for evaluating U-CIT in this work, i.e., highly-configurable systems and event-driven systems.

U-CIT is not a methodology for deciding what needs to be tested. It, in fact, takes as input a set of entities to be covered and aims to cover them in a minimum number of test cases by accommodating as many entities as possible in a single test case. Note that for a given CIT problem, regardless of whether an U-CIT constructor is to be used or a specialized constructor is to be developed, entities to be covered

need to be enumerated and a procedure needs to be devised to determine whether a given set of entities can be covered together in a single test case or not. Once these are given, though, U-CIT provides a constructor right away.

Furthermore, U-CIT does not aim to replace existing CIT approaches. We, indeed, don't see much value in using U-CIT to compute the same CIT objects that the existing CIT constructors compute, as the generalized U-CIT constructors may not be as efficient and as effective as their specialized counterparts. We rather aim to reduce the barriers to applying CIT to other domains and testing problems by generalizing the construction of CIT objects as much as possible, so that the collective effort spent for developing U-CIT constructors can be leveraged to address a wider spectrum of CIT problems.

In this thesis, we present two U-CIT constructors, namely *cover-and-generate* and *generate-and-cover*. While the former aims to cover as many entities as possible in a cluster first and then generates a test case for the cluster, the latter generates a test case first and then marks all the entities accommodated by the test case as covered.

To evaluate U-CIT, we then carry out three case studies, each of which focuses on a different CIT problem. In the first study, we use U-CIT to compute structural code coverage-based test suites. In the second study, we use U-CIT to improve a number of existing order-based covering arrays for testing event-driven systems by taking the reachability constraints imposed by graph-based models directly into account during the construction of CIT objects. In the last study, we use U-CIT to compute usage-based CIT objects, where the entities to be covered are determined according to their usage statistics in the field – an approach which is of importance especially when standard covering arrays are not desirable due to their sizes.

In these studies, we observed that it was either unclear how to use the existing constructors (if at all possible) to compute the requested CIT objects; or the existing constructors required non-trivial modifications or excessive number of test cases to guarantee a full coverage. U-CIT, on the other hand, used the same constructor to compute all the requested CIT objects without requiring any modifications, demonstrating the flexibility of the proposed approach.

We present new method, which we call "hints", to improve the efficiency of U-CIT constructors by capturing the domain knowledge of systems in the forms of hints. The idea behind using hints stems from an observation of ours: Testable entities to be covered are typically composed of the same set of sub-entities, e.g., the same conjuncts appear in multiple testable entities. Therefore, in the processes of computing U-CIT objects, the same constraints are often solved multiple times.

Consequently, capturing the relationships between these recurring constraints (i.e., sub-entities) in the form of hints can improve the efficiency of U-CIT constructors by reducing the number of times the solver is called and/or by calling the solver with simpler constraints.

We also carry out user studies to further evaluate the proposed approach. More specifically, we observe human subjects working on the smaller instances of the very same CIT problems we study in this work and report the results we obtained together with the insights we gained.

In previous work (Mercan & Yilmaz, 2016), we presented an initial set of definitions for U-CIT and provided an algorithm for computing U-CIT objects. And, we did this without providing any implementations or empirical evaluations. In this thesis, however, we present a simplified set of more formal definitions, an additional U-CIT constructor, a tool implementing the U-CIT constructors, and three case studies together with user studies, in which U-CIT is evaluated.

The contributions of this thesis can be summarized as follows:

- A flexible approach, U-CIT, for computing combinatorial objects for testing,
- Two constructors together with a tool implementing these constructors to compute U-CIT objects,
- Definition and construction of structure-based U-CIT objects,
- Definition and construction of order-based U-CIT objects,
- Definition and construction of usage-based U-CIT objects,
- A series of experiments demonstrating the flexibility of U-CIT,
- A new method to improve the efficiency of U-CIT constructors by capturing the domain knowledge of systems in the forms of hints,
- User studies demonstrating the usability of U-CIT.

The remainder of the paper is organized as follows: Chapter 2 presents some background information; Chapter 3 discusses related work; Chapter 4 introduces U-CIT on a motivating example; Chapter 5 develops two constructors for computing U-CIT objects; Chapter 6 presents three case studies, demonstrating the drawbacks of the existing CIT approaches and how U-CIT addresses these drawbacks; Chapter 7 describes hints approach with two case studies; Chapter 9 provides a general discussion of the applicability of U-CIT; Chapter 10 discusses threats to validity; and Chapter 11 presents concluding remarks and possible directions for future work.

2. BACKGROUND INFORMATION

This chapter presents some background information about combinatorial interaction testing as well as its widely used two covering array types, and structural coverage criterion with its well known two different criterion.

2.1 Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) approaches systematically sample the input space and test only the selected instances of the system's behavior (Nie & Leung, 2011; Yilmaz et al., 2014). The term "input" in CIT is used in the most general sense to refer to any factor, which can affect program executions, such as configuration options, input parameters, user events, etc.

CIT approaches typically model the software under test as a set of parameters, each of which takes its values from a discrete domain. As not all possible combinations of parameter values may be valid in practice, the model can also have a set of constraints, which invalidate certain combinations. Based on this model, CIT then generates a sample, i.e., a set of test cases, which from now on will be referred to as a *CIT object*, meeting a specified *coverage criterion*. That is, the sample contains some specified combinations of parameters and their values.

2.1.1 Standard Covering Arrays

A *t-way covering array* is well-known CIT object which is a set of valid configurations, where t is called the *coverage strength* – requires that each valid combination

Table 2.1 An example of 2-way covering array for a system having 5 boolean configuration options and with a system constraint $o_1 \neq o_2$.

o_1	o_2	o_3	o_4	o_5
0	1	1	1	1
1	0	0	0	0
0	1	0	0	1
1	0	1	1	0
0	1	1	0	0
1	0	0	1	1

of parameter values for every combination of t parameters appears at least *once* in the sample (Cohen et al., 1997), aiming to reveal all the failures caused by the interactions of t or fewer parameters.

An example of a 2-way covering array for a system having 5 boolean configuration options $\{o_1, o_2, o_3, o_4, o_5\}$ which either takes 0 or 1, and with a system constraint $o_1 \neq o_2$ which invalidates some certain configurations is given in Table 2.1. Note that every possible combinations of parameter values is covered at least by one of the configurations of given covering array except $< o_1 = 0, o_2 = 0 >$ and $< o_1 = 1, o_2 = 1 >$ due to the given system constraint. For o_2 and o_3 pair (since t=2), for instance, $< o_2 = 1, o_3 = 1 >$, $< o_2 = 0, o_3 = 0 >$, $< o_2 = 1, o_3 = 0 >$, and $< o_2 = 0, o_3 = 1 >$ are covered by the first, second, third, and fourth configurations, respectively.

2.1.2 Sequence Covering Arrays

Sequence covering arrays (Chee, Colbourn, Horsley & Zhou, 2013; Kuhn, Kacker, Lei & others, 2010) are typically used for testing event-driven systems, such as graphical user interfaces (Kuhn, Higdon, Lawrence, Kacker & Lei, 2012a; Yuan et al., 2011), where each symbol corresponds to an event, such as clicking on a button and each row of the array corresponds to a test case. In event driven systems, the system behavior typically depends on the order, in which the events occur. For example, the behavior of a word processor would be quite different when a "paste" event is followed by a "copy" event on an empty clipboard, compared to that of a "copy" event followed by a "paste" event.

In this context, (n, k) sequence covering arrays, aim to exercise the system behavior caused by the orderings of k or fewer events by testing all possible permutations of k distinct events. More formally, an (n, k) sequence covering array (S-CA) is a set of permutations of n distinct symbols, such that each permutation of k distinct symbols appears as a sub-sequence (i.e., not necessarily in a consecutive manner) in

Table 2.2 An example (4, 3) sequence covering array for symbols $\{e_1, e_2, e_3, e_4\}$.

$[e_1, e_4, e_2, e_3]$
$[e_2, e_1, e_3, e_4]$
$[e_2, e_4, e_3, e_1]$
$[e_3, e_1, e_2, e_4]$
$[e_3, e_4, e_2, e_1]$
$[e_4, e_1, e_3, e_2]$

at least one of the permutations included in the set.

An example (4, 3) sequence covering array for the symbols $\{e_1, e_2, e_3, e_4\}$ is given Table 2.2. Note that each row in this table corresponds to a permutation of the 4 symbols and that each permutation of 3 distinct symbols is covered at least once. For example, the first row, i.e., $[e_1, e_4, e_2, e_3]$, covers the permutations $[e_1, e_4, e_3]$, $[e_1, e_2, e_3]$, and $[e_4, e_2, e_3]$.

2.2 Structural Coverage Criterion

Structural coverage criterion (Chilenski & Miller, 1994; Javeed, 2015) are generally studied in two groups: control flow and data flow. Whereas data flow considers the flow of data, i.e., variables definitions and their usage in the codes, the control flow criterion are based on measuring control flow between block of statements. In this work, we study two well know structural coverage criterion: *Decision Coverage* and *Condition Coverage*.

2.2.1 Decision Coverage

Decision coverage (DC) (Chilenski & Miller, 1994) is a structural coverage criterion which states that all possible outcomes ($true$ or $false$) of Boolean expressions at decision points (e.g., if statement, while loop etc.) needs to be exercised. In other words, decision coverage aims to validate all the accessible source code by ensuring that each one of the possible branches from each decision point is executed at least once.

As an example, consider the if statement given in Figure 2.1. The decision condition

```
if ((x == 1) && (y == 2) && (z > 3)) {
   ...
   ...
}
...
```

Figure 2.1 An example of decision condition for an if statement.

Table 2.3 Test cases ensuring full coverage for the if statement given in Figure 2.1 under decision coverage criterion.

test cases			
x	**y**	**z**	**outcome**
1	2	5	$true$
1	3	3	$false$

in the if statement contains 3 integer parameters (x, y and z) each of which can take any possible integer values. To get a full coverage under decision coverage criterion for this code segment, one can use the test cases given in Table 2.3 both covering $true$ and $false$ outcome of the decision.

2.2.2 Condition Coverage

Condition coverage (CC) (Chilenski & Miller, 1994) is another structural coverage criterion which states that all possible outcomes ($true$ or $false$) of every Boolean condition which does not have any logical operators such as AND, OR, or NOT, needs to be tested. As an example, for the if statement given in Figure 2.1, there are 3 conditions: $x == 1$, $y == 2$, and $z > 3$. Therefore, to achieve a full coverage under condition coverage criterion, following Boolean expressions needs to be tested: $x == 1$, $!(x == 1)$, $y == 2$, $!(y == 2)$, $z > 3$, and $!(z > 3)$.

The example test cases given in Table 2.4 satisfies full coverage under condition coverage, i.e., every possible outcome of Boolean conditions is exercised. However, note that, outcome of the both test cases are $false$

Table 2.4 Test cases ensuring full coverage for the if statement given in Figure 2.1 under condition coverage criterion.

test cases			
x	**y**	**z**	**outcome**
2	2	2	$false$
1	1	5	$false$

3. RELATED WORK

This chapter presents some related work on CIT object computation techniques, mainly standard CAs (Chapter 2.1.1) and sequence CAs (Chapter 2.1.2), how the constraints are handled in CIT and usage of seeding mechanism.

3.1 Computing CIT Objects

The results of many empirical studies suggest that a majority of parameter-related failures in practice are often caused by the interactions of only a small number of parameters (Blue, Hicks, Rawlins & Tzoref-Brill, 2019; Li, Chen & Gong, 2019; Nie & Leung, 2011; Petke, Cohen, Harman & Yoo, 2015). That is, t is generally much smaller than the number of parameters, typically $2 \leq t \leq 6$ with t=2 (i.e., pairwise testing) being the most common case (Alazzawi & Rais, 2019; Charbachi, Eklund & Enoiu, 2017; Mohammad & Valepe, 2019). Thus, covering arrays have been successfully used in many domains, including systematic testing of input parameters (Eitner & Wotawa, 2019; Jarman & Smith, 2019; Rao & Li, 2021), software configurations (Mukelabai & Nešić, 2018; Yilmaz, Cohen & Porter, 2006b), software product lines (Lopez-Herrejon, Fischer, Ramler & Egyed, 2015; Qian, Zhang & Wang, 2018), graphical user interfaces (Klammer, Ramler & Stummer, 2016; Michaels, Adamo & Bryce, 2020; Mirzaei & Garcia, 2016), multi-threaded applications (Qi, Tsai, Colbourn, Luo & Zhu, 2018), and network protocols (Choi, 2017). Consequently, efficient and effective ways of computing covering arrays are of great practical importance (Li et al., 2019; Nie & Leung, 2011), which is also evident from many works in the literature (Akhtar & Maity, 2016; Bombarda & Gargantini, 2020; Luo & Lin, 2021; Mercan, Yilmaz & Kaya, 2018; Sheng, Jiang & Wei, 2019; Wu, Nie, Kuo, Leung & Colbourn, 2014; Zhang, Cai & Ji, 2017).

U-CIT is different in that it defines the testable entities to be covered and the space

which the test cases will be selected as a constraint problem. Then, by solving this problem, it computes a CIT object. U-CIT does not need to know about the semantics of constraints. As long as a solver which checks whether given a set of constraints is satisfiable, is provided, it provides the constructor right away.

Furthermore, while these constructors are developed to compute covering arrays, the constructors we have presented in this paper compute U-CIT objects, of which the covering arrays are a special instance, by solving the cov-CSP problem. Note that, U-CIT does not aim to replace existing CIT approaches. We, indeed, don't see much value in using U-CIT to compute the same CIT objects that the existing CIT constructors compute, as the generalized U-CIT constructors may not be as efficient and as effective as their specialized counterparts. We rather aim to reduce the barriers to applying CIT to other domains and testing problems by generalizing the construction of CIT objects as much as possible, so that the collective effort spent for developing U-CIT constructors can be leveraged to address a wider spectrum of CIT problems.

3.2 Constraint Handling in CIT

Using constraint solving techniques to compute covering arrays is not a new idea (Wu, Changhai, Petke, Jia & Harman, 2019). Several approaches empirically demonstrate that ignoring constraints when computing covering arrays often results in a waste of resources (Cohen, Dwyer & Shi, 2008; Lin & Luo, 2015; Yu, Lei, Nourozborazjany, Kacker & Kuhn, 2013). Yilmaz et al. introduce test case-specific constraints (Yilmaz, 2013b), which invalidate option setting combinations on a per test case basis, and demonstrate that not handling them can cause masking effects (Yilmaz, Dumlu, Cohen & Porter, 2014). CASA uses a SAT solver to handle the system-wide constraints within its simulating annealing algorithm (Garvin, Cohen & Dwyer, 2011). Banbara et al. describe several encodings suitable for modern SAT solvers (Banbara, Matsunaka, Tamura & Inoue, 2010). In another work of Banbara et al. (Banbara & Inoue, 2017), they present a new approach to compute CIT objects for constrained systems using Answer Set Programming. Hnich et al. present a SAT encoding designed for incomplete SAT solvers (Hnich, Prestwich & Selensky, 2004; Hnich, Prestwich, Selensky & Smith, 2006). Furthermore, Yan and Zhang (Yan & Zhang, 2006) proposes a SAT-based method with a backtracking mechanism for computing covering arrays. In a more recent work (Jin, Kitamura,

Choi & Tsuchiya, 2018), Jin et al. developed a new algorithm by combining the existing satisfiability based techniques to compute covering arrays.

The constraints, however, in U-CIT are interpreted quite differently than the ones used in existing CIT approaches. More specifically, while the constraints in existing CIT approaches are typically used to specify combinations of parameter values that should be avoided, they are used in U-CIT to specify both the combinations (i.e., the entities) to be covered and the space of valid test cases, from which the samples are drawn. Therefore, the scope of a constraint in existing CIT approaches is all the test cases included in a covering array. That is, all of the selected test cases should satisfy all the constraints. On the other hand, the scope of a constraint representing an entity to be covered in U-CIT is limited to a single test case. That is, such a constraint needs to be satisfied by at least one test case, rather than by all the test cases selected, allowing a considerable amount of flexibility.

For instance, in our running example (Figure 1.1), expressing o_5 and $\neg o_5$ (i.e., the outcomes of the decision in line 6) as constraints to selectively determine what to cover in standard covering arrays, prevents the generation of any covering arrays as these conflicting constraints are enforced to be satisfied by all of the selected test cases. In U-CIT, however, these constraints are required to be satisfied by different test cases. For example, in Figure 1.1d, while the former constraint is satisfied by the first and third test cases, the latter one is satisfied by the second test case.

3.3 Seeding

Seeding has also been frequently used in combinatorial interaction testing (Gladisch, Heinzemann, Herrmann & Woehrle, 2020; Nie & Leung, 2011). Some example uses can be summarized as follows: 1) to guarantee the inclusion of certain configurations by having them in the seed (Deng, Zhang, Li, Yan & Zhang, 2020; Gao, Deng & Yan, 2019); 2) to reduce the cost of testing by including already tested configurations in the seed (Ma, Zhang, Xue, Li, Liu, Zhao & Wang, 2018); and 3) for incremental construction of covering arrays by using lower strength covering arrays as seeds to compute higher strength covering arrays (Galinier, Kpodjedo & Antoniol, 2017; Yilmaz et al., 2014). In this work, we have developed a seeding mechanism for U-CIT and used it in two different ways: 1) to combine multiple coverage criteria and 2) to incrementally construct U-CIT objects (Chapter 6.1).

4. UNIFIED COMBINATORIAL INTERACTION TESTING

In this chapter, we first continue with our running example (Chapter 1) to show how CIT fails or generates an excessive number of test cases to compute a CIT object meeting a specified coverage criterion. Then, we show how U-CIT computes the same requested CIT object in a more flexible way. Finally, we give some formal definitions for U-CIT.

Note that, to increase the readability of this chapter, Figure 1.1 is replicated as Figure 4.1 in this chapter.

4.1 Motivating Example

We provide more details on our running example discussed in Chapter 1. In this example, we are concerned with compile-time configuration options implemented in the form of preprocessor directives, such as `#ifdef` and `#ifndef` directives found in C and C++. Figure 4.1a presents a hypothetical system with 6 compile-time configuration options, namely $o_1, \ldots, o_6$, each of which happens to have two levels of settings *(T)rue* and *(F)alse*. In the remainder of the thesis, we use the term "if-then-else directive" to refer to an `#ifdef`, `#ifndef`, or a similar conditional branch directive, the conditions of which are comprised of only compile-time configuration options and/or constants. Note that such directives allow the decision outcomes to be directly controlled from outside the system by modifying the settings of the compile-time options as a part of the build process.

An if-then-else directive essentially describes how configuration options interact with each other. That is, the outcome of a decision (thus the behavior of the system) may change due to these interactions. Consequently, these interactions may need to be tested. To this end, one structural test adequacy criterion that the developers

```
1  #ifdef (o1 && o2)
2    #ifdef (o3 || o4)
3      ...
4    #endif
5  #endif

6  #ifdef (o5)
7    #ifdef (o6)
8      ...
9    #endif
10 #endif
```

(a)

test cases						decision outcomes			
o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
T	F	T	F	F	F	F	-	F	-
F	T	F	T	T	F	F	-	T	F
T	T	T	T	F	T	T	T	F	-
T	F	F	F	T	F	F	-	T	F
F	F	T	F	T	T	F	-	T	T
F	F	F	T	F	T	F	-	F	-
T	T	T	F	F	T	T	T	F	-

(b)

entities to be covered
$e_1 : (o_1 \wedge o_2)$
$e_2 : \neg(o_1 \wedge o_2)$
$e_3 : (o_1 \wedge o_2) \wedge (o_3 \vee o_4)$
$e_4 : (o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)$
$e_5 : (o_5)$
$e_6 : (\neg o_5)$
$e_7 : (o_5 \wedge o_6)$
$e_8 : (o_5 \wedge \neg o_6)$

(c)

test cases						decision outcomes			
o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
T	T	T	T	T	T	T	T	T	T
F	F	T	F	F	T	F	-	F	T
T	T	F	F	T	F	T	F	T	F

(d)

Figure 4.1 (a) An example set of preprocessor directives for a system with 6 compile-time configuration options, (b) an example 2-way standard covering array created for the system, (c) entities to be covered to obtain full coverage under the decision coverage criterion, and (d) an example test suite obtaining full coverage under the decision coverage criterion.

can use is the decision coverage criterion (Chapter 2.2). A full coverage under DC is obtained when every decision, such as $(o_1 \wedge o_2)$ and $(o_3 \vee o_4)$ in Figure 4.1a, is evaluated to both $true$ and $false$.

Consider a scenario where the goal is to create a DC-adequate test suite for the system given in Figure 4.1a. Note that since a single configuration can cover multiple decision outcomes, the number of configurations required to obtain full coverage under DC can be reduced by covering as many outcomes as possible in each of the selected configurations. This is, indeed, the main motivation behind CIT. Therefore, CIT should be of help.

4.1.1 Applying standard CIT

It, however, turns out that standard covering arrays are infeasible to achieve the aforementioned coverage criterion in an efficient and effective manner.

As an in initial attempt, a standard 2-way covering given in Figure 4.1b is created. The first 6 columns in this figure present the 2-way covering array and the last 4 columns depict the outcomes of the decisions: 'T' for $true$, 'F' for $false$, and '$-$' for decisions that are not exercised due to some unsatisfied guard conditions. For example, the first row indicates that the decision $(o_3 \vee o_4)$ is not exercised by the configuration $(o_1 = T, o_2 = F, o_3 = T, o_4 = F, o_5 = F, o_6 = F)$, because the guard condition $(o_1 \wedge o_2)$ evaluates to F.

This covering array while obtaining a full coverage for the if-then-else directive between the lines 6 and 10 in Figure 4.1a, obtains only 75% DC coverage for the if-then-else directive between the lines 1 and 5, covering 3 out of 4 decision outcomes required for full coverage. More specifically, out of the decision outcomes $\{(o_1 \wedge o_2)$, $\neg(o_1 \wedge o_2)$, $(o_1 \wedge o_2) \wedge (o_3 \vee o_4)$, $(o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)\}$, the last one where the inner decision $(o_3 \vee o_4)$ needs to be evaluated to F, is not covered. Note that this outcome can only be achieved with a single 4-way combination, in which o_1=T, o_2=T, o_3=F, and o_4=F.

One solution approach to overcome this issue is to increase the strength of the covering array, i.e., to use a larger t. This, however, can excessively increase the number of configurations to be tested. For example, since the missing combination in our example is a 4-way combination, to guarantee the coverage of this combination, a 4-way covering array needs to be created at the very least. However, a 4-way covering array for this scenario can have as many as 28 configurations.

An alternative approach is to use a variable-strength covering array, requiring a 4-way coverage only for the options $\{o_1, \ldots, o_4\}$. However, since what is actually being requested is the exhaustive testing of all possible combinations of settings for these 4 binary options, at least 16 configurations are required by this alternative.

Note that decision outcomes that need to be covered cannot be expressed as constraints in standard covering arrays in an attempt to selectively determine what to cover. This is because constraints in standard covering arrays are globally enforced. That is, the constraints should be satisfied by each and every configuration included in the covering array. Therefore, expressing the decision outcomes as constraints in standard covering arrays prevents the creation of any CIT objects because the alternative outcomes of a decision are guaranteed to conflict with each other. For example, since the outcomes of the decision at line 6 in Figure 4.1a, i.e., o_5 and $\neg o_5$, conflict with each other, no configuration satisfying both of these constraints can be

generated; thus, no standard covering array can be constructed.

4.1.2 Applying U-CIT

U-CIT, on the other hand, can flexibly be used as follows to obtain DC-adequate test suites. Each entity to be covered corresponds to a distinct decision outcome. The entities are then expressed as constraints by using Boolean logic where each configuration option is represented by a Boolean variable. For our running example, Figure 4.1c presents all the entities required to be covered to obtain full coverage under the DC criterion. For instance, the first two entities (e_1 and e_2) represent the T and F outcomes of the decision at line 1 in Figure 4.1a, respectively.

Given the entities in Figure 4.1c, an U-CIT constructor divides them into 3 clusters: $\{e_1, e_3, e_5, e_7\}$, $\{e_2, e_6\}$, and $\{e_4, e_8\}$, such that all the constraints within a cluster can be satisfied together and that the number of clusters required to cover all the entities is minimized as much as possible.

Each cluster represents a set of decision outcomes that can be covered together in a single configuration. Therefore, a solution computed for a cluster represents a configuration, which covers all the decision outcomes included in the cluster. Consequently, the U-CIT constructor generates the three configurations (one for each cluster) given in Figure 4.1d, which obtain full coverage under the DC criterion; the first configuration covers the entities $\{e_1, e_3, e_5, e_7\}$, the second configuration covers $\{e_2, e_6\}$, and last configuration covers $\{e_4, e_8\}$.

Note that neither the clusters nor the configurations generated in this study are unique in the sense that there are other sets of configurations that an U-CIT constructor can generate to achieve full coverage. This is indeed similar to what we have in standard covering arrays as different t-way covering arrays can be computed for the same input space model.

Note further that although half of the constraints in Figure 4.1c conflict with the other half, it does not create an issue for U-CIT. This is because as each of these constraints represents an entity to be covered, U-CIT enforces them at the level of a test case. This, in turn, improves the flexibility of CIT, compared to enforcing the constraints at the level of a test suite as is the case with standard covering arrays where each and every test case included in a test suite should satisfy all the constraints. That is, U-CIT aims to satisfy each constraint representing an entity in at least one test case, rather than enforcing all the selected test cases to satisfy

all of the entity constraints. For example, in the test suite given in Figure 4.1d, the constraint for entity $e_2 : \neg(o_1 \wedge o_2)$ is satisfied by the second configuration only. The other configurations included in this test suite, indeed, violate this constraint.

4.2 U-CIT Formal Definitions

U-CIT takes as input a set of entities E to be covered and a model $M =< P, D, C >$, where $P = \{p_1, p_2, \ldots, p_k\}$ is a set of parameters, $D = \{D_1, D_2, \ldots, D_k\}$ is a set of respective domains of values, and C is a constraint defined over P. While C defines the space of valid test cases, from which the samples are drawn, E specifies what needs to be covered by these samples.

Next, we make a number of definitions, starting from the "standard" definitions and going towards the U-CIT-specific ones:

Definition 1 *A* constraint *is a tuple $< P', R >$ where $P' \subseteq P$ is a subset of $l \leq k$ parameters and R is an l-ary relation on the corresponding domains.*

Definition 2 *An* evaluation *is a function from a subset of parameters to a particular set of values in the corresponding subset of domains.*

Definition 3 *An* evaluation *satisfies a constraint $< P', R >$, if the values assigned to the parameters in P', satisfies the relation R.*

Definition 4 *An evaluation is* consistent *with respect to a set of constraints, if it satisfies all the constraints.*

Definition 5 *An evaluation is* complete*, if it includes all the parameters in P.*

Definition 6 *An* U-CIT testable entity *is a constraint over a subset of P, which has at least one evaluation consistent with C, representing an entity to be covered in testing.*

Definition 7 *An* U-CIT test case *is a complete evaluation of P, which is consistent with C.*

Definition 8 *An U-CIT testable entity is said to be* covered *by an U-CIT test case, if and only if the test case is consistent with the testable entity.*

Definition 9 *Given an U-CIT model $M =< P, D, C >$ and a set of testable entities*

E to be covered, an U-CIT object *is a set of U-CIT test cases, such that every U-CIT testable entity in E, is covered by at least one U-CIT test case.*

Going back to our running example in Chapter 4.1, the U-CIT model $M =< P, D, C >$ is defined as follows: $P = \{o_1, \ldots, o_6\}$, $D = \{\{T, F\}, \{T, F\}, \{T, F\}, \{T, F\}, \{T, F\}, \{T, F\}\}$, and $C : true$, indicating that all possible configurations are valid. An U-CIT testable entity corresponds to a distinct decision outcome expressed as a constraint in Boolean logic. The testable entities to be covered $E = \{e_1, \ldots, e_8\}$ are then defined as they are given in Figure 4.1c. For example, the testable entity e_1 is defined as $\neg(o_1 \wedge o_2)$, representing the F outcome of the first decision in Figure 4.1a. An U-CIT test case corresponds to a configuration, in which each configuration option assumes the value of either T or F, such as the second configuration in Figure 4.1d where $o_1 = F$, $o_2 = F$, $o_3 = T$, $o_4 = F$, $o_5 = F$, and $o_6 = T$. An U-CIT object then corresponds to a decision coverage-adequate set of U-CIT test cases, such as the ones given in Figure 4.1d.

5. COMPUTING U-CIT OBJECTS

It turns out that computing U-CIT objects requires us to solve an interesting constraint satisfaction problem, which we call *cov-CSP* (Mercan & Yilmaz, 2016), inspired from the theoretical concepts for "measuring" the level of consistency in paraconsistent logic (i.e., "inconsistency-tolerant" systems of logic) (Makaś, 2016; Rescher & Manor, 1970; Schotch & Jennings, 1980).

Given a set of constraints H, cov-CSP aims to divide H into a minimum number of *satisfiable* clusters. That is, *cov-CSP* seeks to satisfy the constraints, not necessarily as a whole, but in groups. We first define cov-CSP in the most general sense and then show how solving this problem helps compute U-CIT objects.

Definition 10 *Given a set of constraints $H = \{h_1, \ldots, h_m\}$, cov-CSP divides H into a minimum number of clusters $S = \{H_1, \ldots, H_n\}$, such that $\bigcup_{H_i \in S} H_i = H$ and that for each $H_i \in S$, $\bigwedge_{h \in H_i} h$ is satisfiable, i.e., all the constraints in a cluster are satisfiable together.*

Given a model $M =< P, D, C >$ and a set of U-CIT testable entities to be covered $E = \{e_1, \ldots, e_m\}$, each of which is represented as a constraint, computing an U-CIT object proceeds by first solving the cov-CSP problem, so that E is divided into a "minimum" number of satisfiable clusters $S = \{E_1, \ldots, E_n\}$ (as specified by Definition 10). Note that since computing the global minimum may not be computationally feasible (or desirable), U-CIT aims to compute an approximation to it.

Each cluster depicts a set of testable entities that can be tested together. Therefore, a solution for a cluster, represents a U-CIT test case covering all the U-CIT testable entities included in the cluster. Consequently, the collection of all the test cases generated (one per cluster), constitutes a U-CIT object covering each testable entity in E at least once.

The only remaining detail to ensure the generation of valid test cases, is to take the model constraint C into account. To this end, when checking the satisfiability

of a cluster $E_i \in S$ or computing a solution for it, the constraint to satisfy simply becomes $C \wedge \bigwedge_{e \in E_i} e$.

Note that, in order to reduce the number of test cases required, it is desirable to avoid redundancy as much as possible by covering each testable entity in exactly one test case. However, a testable entity, in the process of covering other testable entities, may end up being covered by multiple test cases. This can happen unintentionally (i.e., by chance) or intentionally to satisfy the model constraint C.

Next, we present two constructors for computing U-CIT objects (thus, for solving the cov-CSP problem), namely *cover-and-generate* and *generate-and-cover.*

5.1 The Cover-and-Generate Constructor

Algorithm 1 The cover-and-generate constructor for computing U-CIT objects

Input: A test space model $M =< P, D, C >$
Input: A set of testable entities E to be covered
Output: An U-CIT object T

```
1:  S ← {}
2:  for each testable entity e ∈ E do
3:      accommodated ← false
4:      for each E' ∈ S do
5:          if satisfiable(e ∧ ⋀_{e'∈E'} e' ∧ C) then
6:              E' ← E' ∪ {e}
7:              accommodated ← true
8:              break
9:          end if
10:     end for
11:     if not accommodated then
12:         S ← S ∪ {{e}}
13:     end if
14: end for
15:
16: T ← {}
17: for each E' ∈ S do
18:     T ← T ∪ solve(C ∧ ⋀_{e'∈E'} e')
19: end for
20: return T
```

The cover-and-generate constructor (Algorithm 1) maintains a pool S of clusters, each representing a set of testable entities that can be covered together. The pool is initially empty (line 1). Then, for each testable entity $e \in E$, we attempt to accommodate it in an existing cluster $E' \in S$ (lines 4-10). To this end, we check to see if e is satisfiable together with all the constraints in E' as well as with the model constraint C, i.e., whether $e \wedge \bigwedge_{e' \in E'} e' \wedge C$ is satisfiable (line 5). If so, e is included in E' (line 6), indicating that e can be accommodated together in a single test case with the other testable entities in E'. Otherwise (i.e., if no such cluster is found), we populate S with a new cluster initially having only e (line 12). Once all the testable entities are processed, for each cluster $E' \in S$, we generate a test case by solving $C \wedge \bigwedge_{e' \in E'} e'$ (line 18). The collection of all the test cases generated (T), is then returned as the U-CIT object computed, covering all the testable entities in E (lines 17-20).

5.2 The Generate-and-Cover Constructor

The generate-and-cover constructor associates a cluster with an U-CIT test case, rather than with a set of U-CIT testable entities. Conceptually, this constructor generates a test case first and then marks all the testable entities accommodated by the test case as covered. Therefore, it is different than the cover-and-generate constructor, which attempts to cover as many testable entities as possible in a cluster before generating a test case. Consequently, the set of clusters maintained through the iterations of the generate-and-cover constructor, simply represents the U-CIT test cases that have already been included in the U-CIT object being computed.

Given a model $M =< P, D, C >$ and a set of testable entities E to be covered, one way to generate a test case is to compute a solution for the model constraint C, regardless of E. However, generating test cases without taking the testable entities to be covered into account, may make it quite difficult to cover the entities that are hard to cover by chance. We, therefore, employ an alternative approach in this work, which guarantees that at least one previously uncovered testable entity is covered by every test case generated.

Algorithm 2 presents the generate-and-cover constructor. The U-CIT object T is initially empty (line 1). Then, for each testable entity $e \in E$, we check to see if e has already been covered by a test case $t \in T$ (lines 4-9), i.e., if there exists a test case

Algorithm 2 The generate-and-cover constructor for computing U-CIT objects

Input: A test space model $M =< P, D, C >$
Input: A set of testable entities E to be covered
Output: An U-CIT object T

```
1: T ← {}
2: for each testable entity e ∈ E do
3:     accommodated ← false
4:     for each t ∈ T do
5:         if satisfiable(e ∧ t ∧ C) then
6:             accommodated ← true
7:             break
8:         end if
9:     end for
10:    if not accommodated then
11:        T ← T ∪ solve(e ∧ C)
12:    end if
13: end for
14: return T
```

$t \in T$, which is consistent with e (line 5). If no such test case is found, a new U-CIT test case covering e, is generated by solving the constraint $e \wedge C$ and T is populated with the newly generated test case (lines 10-12). Once all the testable entities in E have been processed, T is returned as the U-CIT object computed (line 14).

5.3 A Seeding Mechanism

Both of the constructors we have discussed so far can also take as input a *seed*, which in this context refers to a set of U-CIT test cases. Given a seed, all the U-CIT testable entities in the seed, are considered to have already been covered and additional U-CIT test cases are generated only to cover the remaining entities.

To this end, the only change that needs to be made is to modify line 1 in Algorithms 1 and 2, such that instead of starting with an empty pool of clusters, we start with an initially populated pool of clusters, each of which is created to include a single U-CIT test case in the seed. Nothing else in the algorithms needs to be changed.

In Chapter 6.1, we use the seeding mechanism both to compute higher strength U-CIT objects from lower strength U-CIT objects (by using the lower strength objects as seeds) and to generate U-CIT objects that satisfy multiple coverage criteria (by

Table 5.1 An U-CIT object (second column) created for the set of satisfiable clusters $S = \{E_1, E_2, E_3\}$ (first column) obtained for the testable entities in Figure 4.1c.

satisfiable clusters	DC-adequate U-CIT object					
$S = \{E_1, E_2, E_3\}$	o_1	o_2	o_3	o_4	o_5	o_6
$E_1 = \{e_1, e_3, e_5, e_7\}$	T	T	T	T	T	T
$E_2 = \{e_2, e_6\}$	F	F	T	F	F	T
$E_3 = \{e_4, e_8\}$	T	T	F	F	T	F

using an object satisfying a coverage criterion as a seed to compute another object satisfying a different coverage criterion).

5.4 Example: Computing DC-Adequate Test Suites as U-CIT Objects

In this section, for illustrative purposes, we use the cover-and-generate constructor (Algorithm 1) to compute DC-Adequate test suites as U-CIT objects using our running example in Chapter 4.1. For the sake of the discussion, however, we introduce the following system-wide constraint to the problem: $(o_2 = F) \implies (o_6 = T)$, i.e., if o_2 is $false$, then o_6 must be $true$, invalidating the combination $(o_2 = F, o_6 = F)$.

Modeling. The U-CIT model is defined as $M =< P, D, C >$, where $P = \{o_1, \ldots, o_6\}$, $D = \{\{T, F\}, \ldots, \{T, F\}\}$, and $C : (\neg o_2 \implies o_6)$. Each U-CIT testable entity then naturally corresponds to a decision outcome to be covered. Figure 4.1c presents all the U-CIT testable entities that need to be covered to obtain full coverage under the decision coverage criterion.

Assuming that the testable entities in Figure 4.1c are processed in the order $e_1, \ldots, e_8$, the cover-and-generate constructor proceeds as follows: First, $e_1 : (o_1 \wedge o_2)$ is processed. Since the pool S is initially empty (line 1), a new cluster $E_1 = \{e_1\}$ is created and S is populated with E_1, i.e., $S = \{E_1\}$ (line 12). Then, $e_2 : \neg(o_1 \wedge o_2)$ is processed. Since $e_1 \wedge e_2 \wedge C$, i.e., $(o_1 \wedge o_2) \wedge \neg(o_1 \wedge o_2) \wedge (\neg o_2 \implies o_6)$, is not satisfiable (line 5), e_2 cannot be placed in E_1. So, a new cluster $E_2 = \{e_2\}$ is created and S is updated to $\{E_1, E_2\}$ (line 12). Next, $e_3 : (o_1 \wedge o_2) \wedge (o_3 \vee o_4)$ is processed. Since $e_1 \wedge e_3 \wedge C$, i.e., $(o_1 \wedge o_2) \wedge ((o_1 \wedge o_2) \wedge (o_3 \vee o_4)) \wedge (\neg o_2 \implies o_6)$, is satisfiable (line 5), e_3 is included in E_1 (line 6). After processing all the remaining testable entities in Figure 4.1c, we have the clusters given in the first column of Table 5.1.

For each cluster in $S = \{E_1, E_2, E_3\}$, we then generate an U-CIT test case by sat-

isfying the constraints included in the cluster together with the model constraint C (lines 16-19). For example, for E_1, solving $e_1 \wedge e_3 \wedge e_5 \wedge e_7 \wedge C$ produces the test case $(o_1 = T, o_2 = T, o_3 = T, o_4 = T, o_5 = T, o_6 = T)$. Processing all the clusters would then generate the U-CIT object given in the second column of Table 5.1 (line 20), which is, indeed, DC-adequate.

5.5 Discussion

Regarding constraints and solvers. The terms "constraint" and "solver" are used in the general sense in U-CIT. That is, any restriction, independent of the logic in which it is specified, is considered to be a constraint and a solver conceptually determines whether a given set of testable entities can be covered together in a single test case or not. Therefore, U-CIT expects that the underlying solver supports essentially a single computational primitive, namely *solve*. The other primitive used in Algorithms 1 and 2, namely $satisfiable$, can actually be implemented by using *solve* as the absence of a solution indicates unsatisfiability.

Having a simple interface between U-CIT constructors and solvers further improves the flexibility of U-CIT. For example, all of the widely-used SAT and CSP solvers, in one form or another, provide a *solve* primitive. Furthermore, this feature also allows application- and domain-specific solvers to be used with U-CIT constructors (Chapter 6.3).

This interface can indeed be further generalized by having *solve* to take as input a set of constraints, each of which can represent a testable entity, a model constraint, or a test case. Since an U-CIT constructor does not then need to interpret these constraints, the testable entities, the model constraints, and the test cases can be expressed in any form desired, which may not even need to be formal.

Regarding constructors. We have presented two constructors in this chapter, namely the cover-and-generate constructor and the generate-and-cover constructor. We introduced the latter solely to mimic one of the simplest ways of generating U-CIT objects: Keep on generating valid test cases until all the required entities have been covered. As such, we use this constructor as a base line for comparisons in our experiments (Chapter 6), demonstrating that computing U-CIT objects in an efficient and effective manner is not trivial. Indeed, the results of our experiments strongly suggest that the cover-and-generate constructor performed better than the

generate-and-cover constructor in reducing both the sizes and the construction times of U-CIT objects (Chapter 6).

We, therefore, generally suggest to use the cover-and-generate constructor. However, the generate-and-cover constructor can still be of practical interest in scenarios especially when it is costly to determine whether multiple testable entities can be covered together or not (due to, for example, the complexity of the constraints to be solved) and when it is easy to cover the entities by chance in valid test cases. Note that the presence of these factors favors the generate-and-cover constructor as multiple testable entities can be covered by generating a valid test case. Furthermore, by making sure that each test case covers at least one previously uncovered testable entity, the generate-and-cover constructor guarantees the convergence into full coverage. Clearly, the end-users can always experiment with both constructors to determine the one to use in their projects.

With all these in mind, we have implemented the U-CIT constructors given in Algorithms 1 and 2 in Python in the form of an extensible tool that can work with any types of constraints and solvers. The tool can be downloaded at `https://github.com/susoftgroup/UCIT/`.

The efficiency and effectiveness of the U-CIT constructors we introduced in this work (i.e., the construction times and the sizes of the U-CIT objects computed), can be effected by the order, in which the testable entities are processed. In the presence of some knowledge regarding a favorable order (or a partial order), the testable entities can be sorted accordingly before they are fed to an U-CIT constructor. If not, a random order can be used by shuffling the entities. Furthermore, the construction process can be repeated multiple times in an attempt to compute smaller U-CIT objects at the cost of increased construction times. In Chapter 6.3.4, we carry out additional set of experiments to evaluate the sensitivity of the cover-and-generate constructor (which generally performed better than the generate-and-cover constructor) to the order the testable entities are processed.

Furthermore, U-CIT constructors may not be as efficient as their specialized counterparts. Our ultimate goal, however, is not to perform better than the existing constructors when U-CIT is used to compute the same CIT objects that these constructors are specifically designed to compute. As a matter of fact, we don't see much value in using U-CIT in such scenarios unless the U-CIT constructors perform better than the existing ones. Our goal is rather to improve the flexibility, thus the applicability, of CIT by eliminating the necessity of developing specialized constructors for every distinct CIT problem, which is not addressed by the existing constructors.

6. EXPERIMENTS

U-CIT does not aim to replace existing CIT constructors, but rather to reduce the barriers to applying CIT to other domains and problems. Note that, in this context, changing the underlying CIT problem is not the same as simply changing the parameters of an existing problem, but rather changing the problem itself. For example, for standard covering arrays, we do not consider the changes in system-wide constraints and/or the changes in model parameters to be a change in the underlying CIT problem. This is because the only thing that changes in such situations is the problem parameters, while the original problem remains intact, which is to cover all valid t-tuples at least once.

To evaluate U-CIT, we, therefore, carry out three case studies, each of which focuses on a different CIT problem. In the first study (Chapter 6.1), we compute structure-based CIT objects to obtain decision coverage-adequate objects. In the second study (Chapter 6.2), we compute order-based CIT objects, where the reachability constraints imposed by an underlying graph-based model are taken into account to cover various sequences of events. In the third study (Chapter 6.3), we compute usage-based CIT objects by selecting the tuples to be covered based on their usage statistics in the field, which is especially useful when standard covering arrays are not desirable due to their sizes.

In each study, we first introduce the CIT problem of interest and discuss the motivation behind this problem. We then discuss and empirically demonstrate that to compute the requested CIT objects, the existing constructors (as they are) require excessive number of test cases to guarantee full coverage. Or, they require non-trivial modifications. Or, it is not clear (if at all possible) how to modify them.We finally express the CIT problems in U-CIT and show that the very same U-CIT constructor (thus, the same construction approach) can compute all of the requested CIT objects in all the studies without any modifications, demonstrating the flexibility of the proposed approach.

In the experiments, we integrate different "solvers" with U-CIT. This, however, is

solely for the purpose of demonstrating that U-CIT can work with different solvers. The very same solver, such as the CSP solver we use in Chapter 6.1, can indeed be used in all the studies.

Note further that although the CIT problems in our studies are different than the ones addressed by existing CIT constructors, we opt to use existing constructors for comparisons in the experiments to justify the need for U-CIT. That is, in these studies, we are not claiming that U-CIT constructors perform better than standard CIT constructors (because the underlying CIT problems are different), but rather demonstrating that a different CIT constructor is indeed needed to compute the requested CIT objects in an efficient and effective manner. Otherwise, i.e., had the existing constructors addressed the CIT problems presented in this paper in an efficient and effective manner, there would be no need for U-CIT.

We, furthermore, use our generate-and-cover U-CIT constructor as a base line to show that computing U-CIT objects is not trivial at all and that better construction approaches, such as the cover-and-generate approach, are needed.

The raw data we obtained from the experiments can be found at `https://github.com/susoftgroup/UCIT/`.

6.1 Structure-Based CIT

In this study, we use the same CIT problem discussed in Chapter 4.1.

6.1.1 Coverage criterion

In (Javeed, 2015; Javeed & Yilmaz, 2015), a novel CIT object has been introduced, which given a structural coverage criterion, such as decision coverage (DC), computes a "minimal" test suite to obtain full coverage under the criterion. In this work, we not only express the same coverage criterion using U-CIT, demonstrating the expressiveness of U-CIT, but also generalize the aforementioned coverage criterion to higher coverage strengths, demonstrating the flexibility of U-CIT. We call this *structure-based CIT*.

In a nutshell, structure-based CIT takes as input the source code of the system under test, a coverage strength t, and a structural code coverage criterion. First, for each outer-most if-then-else directive in the implementation, a *virtual configuration option* is defined. Then, for a given a virtual configuration option, conditions that must be satisfied to obtain a full coverage under the given structural coverage criterion for the respective if-then-else directive, are defined as *virtual settings*. Finally, a number of configurations are selected to cover all valid t-way combinations of virtual option settings. The smaller the number of configurations selected, the better the approach is.

Next, without losing generality, we provide more details by using DC as the structural code coverage criterion of interest. The proposed approach, on the other hand, is readily available to use with other structural coverage criteria, such as condition coverage (Yu & Lau, 2006).

Definition 11 *A* virtual configuration option *(or* virtual option*, in short) represents an outer-most if-then-else directive, which is not nested in another if-then-else directive.*

For example, the system in Figure 1.1a has two virtual options: vo_1 representing the outer-most if-then-else directive between lines 1 and 5 and vo_2 representing the outer-most if-then-else directive between lines 6 and 10.

Definition 12 *Given a virtual configuration option, each feasible outcome of every decision in the respective if-then-else directive, is defined as a* virtual setting *and expressed as a constraint, such that covering all of these virtual settings obtains a full coverage under DC.*

For instance, the virtual option vo_1 in our running example has four virtual settings: $\{o_1 \wedge o_2,\ \neg(o_1 \wedge o_2),\ (o_1 \wedge o_2) \wedge (o_3 \vee o_4),\ (o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)\}$. The first two settings are respectively for covering the $true$ and $false$ branches of the decision $o_1 \wedge o_2$ and the last two settings are respectively for covering the $true$ and $false$ branches of the decision $o_3 \vee o_4$ while taking the guard condition $o_1 \wedge o_2$ into account. Similarly, vo_2 has four virtual settings: $\{o_5,\ \neg o_5,\ o_5 \wedge o_6,\ o_5 \wedge \neg o_6\}$.

Not all virtual settings of a virtual option may be valid due to some conflicting settings required for the actual configuration options that appear multiple times in the same if-then-else directive. Since each virtual setting is expressed as a constraint, an invalid virtual setting can be marked and filtered out by determining whether or not the respective constraint is satisfiable. That is, a virtual setting is invalid, if the respective constraint is not satisfiable. Clearly, covering invalid virtual settings is not required to achieve full coverage. Consequently, in the remainder of the paper,

Table 6.1 Information about the subject applications used in Study 1.

sut	version	description	actual options	virtual options	valid 1-combs	valid 2-combs	valid 3-combs
mpsolve	2.2	Mathematical solver	14	4	30	296	1104
dia	0.96.1	Diagramming application	15	11	42	734	7170
irissi	0.8.13	IRC client	30	11	70	2102	36056
xterm	2.4.3	Terminal emulator	38	31	78	2871	66497
parrot	0.9.1	Virtual machine	51	29	152	10359	426194
gimp	3.2.5	Vector graphics editor	79	28	198	16438	794050
pidgin	2.4.0	IM	53	43	199	17857	986926
python	2.6.4	Programming language	68	49	210	21180	1368012
xfig	2.6.8	Graphics manipulator	79	48	237	26985	1969006
vim	7.3	Text editor	79	49	239	27442	2019176
sylpheed	2.6.0	E-mail client	84	48	258	31597	2451586
cherokee	1.0.2	Web server	97	28	272	32530	2318986

the term "virtual setting" is used to refer to valid virtual settings.

Definition 13 *A* t-combination *is a combination of virtual settings for a combination of* t *distinct virtual options, which is expressed by joining the respective constraints with the AND logical operator.*

As was the case with virtual settings, a t-combination is invalid, if the respective constraint is not satisfiable. In the remainder of the paper, the term "t-combination" is used to refer to valid t-combinations.

Note that each t-combination represents an interaction that can be tested. Going back to our running example and considering that $t = 2$, some example 2-combinations for the virtual options vo_1 and vo_2 are: $(o_1 \wedge o_2) \wedge (o_5)$, testing the interaction between the *true* branches of the decisions at lines 1 and 6; and $((o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)) \wedge (o_6)$, testing the interaction between the $false$ branch of the decision at line 2 and the *true* branch of the decision at line 7.

Definition 14 *Given a set of virtual configuration options, their virtual settings, and a coverage strength* t*,* t-way structure-based coverage criterion K_{struct} *marks all valid* t*-combinations for coverage.*

Definition 15 *Given a set of virtual configuration options, their virtual settings, and a coverage strength* t*, a* t-way structure-based U-CIT object *is a set of actual system configurations, in which each* t*-combination selected by* K_{struct} *is covered by at least one configuration.*

In this context, an actual system configuration is said to cover a t-combination, if the configuration is consistent with the respective constraint.

Note further that the coverage strength t in K_{struct} can be 1, which simply marks the virtual settings of all the virtual options for coverage. Therefore, covering all

valid 1-combinations (i.e., all virtual settings) guarantees to obtain full coverage under DC. Consequently, 1-way structure-based U-CIT objects are the same/similar combinatorial objects we introduced in our short paper (Javeed & Yilmaz, 2015), but expressed in U-CIT, demonstrating the expressiveness of U-CIT.

One issue with the 1-way structure-based U-CIT objects, however, is that they don't take the interactions between structurally isolated if-then-else directives into account. Take the 1-way structure-based object given in Figure 1.1d as an example, although a DC-adequate test suite, it does not, for example, test the interaction between the $true$ branch of the decision $o_1 \wedge o_2$ (line 1) and the $false$ branch of the decision o_5 (line 6).

This issue, which was not addressed in (Javeed & Yilmaz, 2015), can now easily be handled in U-CIT by simply increasing the strength of K_{struct}, demonstrating the flexibility of U-CIT by generalizing the coverage criterion introduced in (Javeed & Yilmaz, 2015). Going back to our running example in Figure 1.1 and considering that $t = 2$, K_{struct} selects $4 * 4 = 16$ 2-combinations for vo_1 and vo_2, covering all the pairwise interactions between the settings of these virtual options.

6.1.2 Study setup

For the evaluations, we used 12 subject applications. Each application had a number of binary compile-time configuration options implemented by using preprocessor directives. Table 6.1 provides information about these subject applications. The columns of this table respectively present the subject applications, their versions and descriptions, the numbers of actual compile-time options they have, the numbers of virtual options extracted, and the numbers of 1-, 2- and 3-combinations selected by our structure-based coverage criterion. Note that since we were not aware of any inter-option constraints for these subject applications, all possible combinations of option settings were considered to be valid. Furthermore, to give an idea about the structural complexities of the virtual options we extracted, Table 6.2 presents the percentages of the virtual options that are of cyclomatic complexities of 2, 3, 4, 5, and ≥ 6, respectively. Throughout the paper cyclomatic complexities are computed on a per virtual option basis by using Radon (Lacchia, 2018) – a tool to compute various code metrics.

All the experiments, unless otherwise stated, were repeated 5 times and carried out on Google Cloud using Intel Xeon CPU 2.30GHz machine with 4 GB of RAM,

Table 6.2 Percentages of the if-then-else directives (one per virtual option) that are of cyclomatic complexity 2, 3, 4, 5, and ≥ 6.

	cyclomatic complexity				
sut	2	3	4	5	≥ 6
mpsolve	0	50	0	0	50
dia	9.09	63.64	27.27	0	0
irissi	0	36.36	36.36	0	27.27
xterm	54.84	25.81	6.45	6.45	6.45
parrot	24.14	37.93	13.79	6.90	17.24
gimp	0	57.14	10.71	28.57	3.57
pidgin	2.33	53.49	25.58	9.30	9.30
python	8.16	63.27	16.33	4.08	8.16
xfig	2.08	50	20.83	14.58	12.50
vim	4.08	48.98	20.41	14.29	12.24
sylpheed	10.42	56.25	8.33	6.25	18.75
cherokee	3.57	32.14	14.29	7.14	42.86

running 64-bit Ubuntu 17.10 as the operating system.

6.1.3 Applying standard CIT

Modeling. The very first observation we make is that standard covering arrays cannot be used (as they are) with virtual options because the settings of virtual options are constraints, rather than discrete values as is the case with standard covering arrays. For example, one setting for vo_1 is $(o_1 \wedge o_2) \wedge (o_3 \vee o_4)$ and another is $(o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)$. To the best of our knowledge, there is no standard covering array constructor that can take constraints as settings. Note that these virtual settings cannot be expressed as constraints in standard constructors either, because such constraints are globally enforced and virtual settings can conflict with each other, which prevents the creation of any covering arrays (Chapter 4.1).

An alternative approach can be to create a standard covering array for the actual configuration options to obtain full coverage under K_{struct}. This, however, may unnecessarily increase the number of configurations required. For example, the standard 2-way covering array given in Figure 1.1b obtains only 38% coverage under the 2-way K_{struct} criterion (covering only 9 out of 24 2-combinations). Since the maximum number of actual configuration options involved in a 2-combination is 6 in this example, a 6-way covering array needs to be used to guarantee full coverage. This, however, is the same as exhaustive testing. Indeed, using variable strength covering arrays as an alternative, also suffers from the same issue.

Table 6.3 Percentages of the 1-, 2-, and 3-combinations covered by standard 2- and 3-way covering arrays.

sut	standard 2-way CA			standard 3-way CA		
	% of t-combinations covered			% of t-combinations covered		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
mpsolve	100	55	23	100	83	56
dia	99	39	18	100	46	27
irissi	100	36	11	100	49	22
xterm	97	49	29	98	55	38
parrot	90	29	8	94	33	15
gimp	95	36	14	98	47	21
pidgin	99	23	11	100	25	17
python	98	31	12	99	36	18
xfig	99	31	12	100	35	18
vim	99	30	11	100	34	18
sylpheed	97	39	16	98	45	25
cherokee	99	21	5	100	28	10

Table 6.4 Percentages of valid 1-combinations of various cyclomatic complexities covered by standard t-way covering arrays.

cyclomatic complexity	standard t-way covering arrays	
	$t=2$	$t=3$
2	100.00	100.00
3	100.00	100.00
4	98.96	100.00
5	98.17	99.84
≥ 6	94.17	97.28

Next, to demonstrate that the CIT problem defined in this study is indeed different than the ones addressed by standard covering arrays, which justifies the need for a different constructor to guarantee full coverage in an efficient and effective manner, we apply standard CIT on the subject applications in Table 6.1.

Evaluations. We first observed that since standard covering arrays do not necessarily take the complex interactions between configuration options into account, they, especially in the presence of tangled options, either fail to obtain full decision coverage or require excessive number of test cases (Javeed, 2015; Javeed & Yilmaz, 2015).

More specifically, we first created standard 2-way and 3-way covering arrays for our subject applications and measured the t-way structure-based coverage they provided for $t = 1$, 2, and 3. The experiments for $t = 1$ and 2 were repeated 30 times, whereas those for $t = 3$ were repeated 5 times as measuring the coverage for higher strengths was costly. The average sizes of the standard 2-way and 3-way covering arrays created were 13.74 and 36.78, respectively.

Table 6.5 Using standard covering arrays to guarantee full coverage under structure-based coverage criterion. The columns indicate the subject application, the coverage strength of the standard covering array computed together with the average construction time and size obtained by repeating the experiments 3 times for 1-, 2-, and 3-way structure-based CIT, respectively. The symbol '-' marks experimental setups, for which the standard constructor failed with an "out of memory" exception.

sut	t-way standard covering arrays created for structure-based CIT								
	1-way structure-based CIT			2-way structure-based CIT			3-way structure based CIT		
	t	time	size	t	time	size	t	time	size
mpsolve	2	0.34	10	4	0.33	54	6	0.56	272
dia	3	0.36	26	5	0.46	134	7	0.97	608
irissi	4	0.90	82	7	-	-	9	-	-
xterm	9	-	-	12	-	-	15	-	-
parrot	10	-	-	15	-	-	18	-	-
xfig	6	-	-	9	-	-	12	-	-
python	5	616.80	299	9	-	-	12	-	-
pidgin	8	-	-	11	-	-	14	-	-
gimp	5	-	-	10	-	-	15	-	-
vim	5	-	-	10	-	-	15	-	-
sylpheed	10	-	-	16	-	-	20	-	-
cherokee	4	73.77	130	7	-	-	10	-	-

Standard covering arrays did not even guarantee DC adequacy, i.e., 1-way structure-based coverage (Table 6.3). More specifically, in about 58% (14 out of 24) of the experimental setups, standard covering arrays could not obtain full DC coverage. Overall, the DC coverages achieved were 97.58% and 99.08%, on average, for $t = 2$ and 3, respectively.

Furthermore, the higher the strength of the structure-based criterion, the more the required combinations were missing from the standard covering arrays (Table 6.3). Overall, the 2- and 3-way standard covering arrays, while respectively covering 34.92% and 43.00% of all the 2-combinations, achieved 14.17% and 23.75% coverage of the 3-combinations.

Similarly, the more the cyclomatic complexity of the virtual options, the more the required combinations were missing (Table 6.4). For example, standard 2-way covering arrays, on average, covered 100.00%, 100.00%, 98.96%, 98.17%, and 94.17% of the 1-combinations for the virtual options with cyclomatic complexities of 2, 3, 4, 5, and ≥ 6, respectively.

We have then created higher strength as well as variable strength covering arrays. For the former, we determined the maximum number of distinct configuration options that appear in a t-way virtual option combination and used it as the strength of the standard covering array. For the latter, we determined the number of distinct

Table 6.6 Using variable strength covering arrays to guarantee full coverage under structure-based coverage criterion. The columns indicate the subject application and the average construction time and size of the variable strength covering arrays computed for 1-, 2-, and 3-way structure-based CIT, respectively. The experiments were repeated 3 times. The symbol '-' marks experimental setups, for which the standard constructor failed with an "out of memory" exception.

sut	variable strength covering arrays created for structure-based CIT					
	1-way structure-based CIT		2-way structure-based CIT		3-way structure-based CIT	
	time	size	time	size	time	size
mpsolve	0.29	8	0.41	47	0.88	252
dia	0.32	8	0.42	48	0.79	202
irissi	0.33	16	0.99	323	554.99	3217
xterm	0.54	512	12.05	4187	-	-
parrot	5.49	3750	-	-	-	-
xfig	364.78	585	-	-	-	-
python	0.40	32	4.70	845	6319.56	13350
pidgin	0.44	256	15.90	3447	-	-
gimp	0.41	32	6.07	730	4317.48	8908
vim	0.41	36	4.82	718	43198.74	9037
sylpheed	19.62	5062	-	-	-	-
cherokee	0.43	18	-	-	-	-

configuration options that appear in each t-way virtual option combination and used it as the coverage strength to be satisfied for these configuration options. All of the covering arrays in these experiments were computed by using `ACTS` (Yu, Lei, Kacker & Kuhn, 2013) and the experiments were repeated 3 times.

Tables 6.5 and 6.6 present the results we obtained. In 75% (27 out of 36) of the experimental setups for computing fixed-strength covering arrays and in 28% (10 out of 36) of the experimental setups for computing variable strength covering arrays, the standard constructor (ACTS) failed with an "out of memory" exception. The tables, therefore, present only the experiments, in which we were able to compute a covering array using the standard constructor. Although the covering arrays we could compute achieved full coverage, they did so at the expense of excessive number of configurations. For comparisons, the reader can refer to Table 6.7 to check the sizes of the U-CIT objects computed for the study.

6.1.4 Applying U-CIT

Modeling. We have defined the U-CIT model as $M =< P, D, C >$, where P is the set of variables representing the actual configuration options; D is their respective domains, i.e., the settings that the actual configuration options can take on; and C

Table 6.7 Information about the structure-based U-CIT objects created. The symbol '*' marks the experimental setups, in which the generate-and-cover constructor timed out after six days.

	1-way				2-way				3-way			
	generate-and-cover		cover-and-generate		generate-and-cover		cover-and-generate		generate-and-cover		cover-and-generate	
sut	time	size	time	size	time	size	time	size	time	size	time	size
mpsolve	0.37	3.00	0.31	3.00	17.61	15.20	2.07	14.00	221.54	93.40	11.99	39.80
dia	0.37	4.40	0.34	4.20	16.35	19.60	2.26	19.40	482.35	131.80	24.79	70.60
irissi	0.69	4.00	0.66	4.00	74.21	25.20	13.16	24.20	8461.64	316.40	139.32	109.20
xterm	0.61	4.20	0.58	4.20	50.54	19.80	5.74	21.20	7025.89	271.60	92.54	79.00
parrot	2.03	10.00	1.95	10.00	877.18	57.80	46.65	55.80	206682.44	841.33	1070.67	317.40
gimp	2.45	8.20	2.27	8.00	825.78	49.80	67.11	48.00	457184.81	998.50	1645.61	272.80
pidgin	2.26	4.40	2.29	4.40	788.98	34.00	31.82	33.40	*	*	628.75	172.00
python	2.16	4.80	2.07	4.40	743.89	36.00	28.68	34.60	*	*	932.46	187.00
xfig	2.81	5.80	2.74	6.00	1355.77	46.00	78.54	45.80	*	*	2311.84	270.00
vim	2.82	6.40	2.69	6.20	1357.64	48.60	56.47	48.60	*	*	1679.70	291.20
sylpheed	3.18	6.00	3.04	6.60	1737.00	49.20	78.20	47.40	*	*	2724.60	279.20
cherokee	3.59	5.00	3.53	5.00	2792.24	45.40	79.89	45.00	*	*	2095.94	252.40

is the model constraint (if any) invalidating certain combinations of option settings. Each U-CIT testable entity then naturally corresponded to a valid t-combination to be covered (Definition 13) and each U-CIT test case naturally corresponded to a configuration, in which every actual configuration option has a valid setting.

We have also used the seeding mechanism of U-CIT in this study to combine multiple coverage criteria. In particular, to construct 1-way structure-based U-CIT objects in some experiments, we used standard 2-way or 3-way covering arrays computed for the actual configuration options, as seeds. By doing so, we effectively computed t-way DC-adequate covering arrays, which not only covered all t-way combinations of actual option settings, but also achieved DC adequacy.

To further demonstrate that the very same seeding mechanism can also be used to incrementally compute U-CIT objects – a well-known approach for computing standard covering arrays (Fouché, Cohen & Porter, 2009), we have used lower strength structure-based U-CIT objects as seeds to compute higher strength U-CIT objects.

Cost. To extract virtual options from source code, we used `cppstats`, which is a static analysis tool for analyzing C/C++ preprocessor-based variability in highly configurable systems (Liebig, Apel, Lengauer, Kästner & Schulze, 2010). The tool parsed the if-then-else directives into an XML-based tree representation. We then simply traversed the representation to identify the elements that corresponded to virtual options. An if-then-else directive, which was not structurally contained in another if-then-else directive simply became a virtual option. Once a virtual option was found, we traversed the respective tree to determine the virtual settings, i.e., visiting the decisions in the possibly nested if-then-else directive. For each decision d with a guard condition g, two virtual settings were created: $g \wedge d$ and $g \wedge \neg d$. All

told, developing a generic script to carry out these steps took about 10 hours.

We have integrated our constructors given in Algorithms 1 and 2 with SATisPy (László, 2018), which is a Python library that interfaces with various SAT solvers, such as MiniSat (Eén & Sörensson, 2003). Since the decisions in the source code were already expressed as Boolean expressions and since the virtual settings (thus, the testable entities) were simply obtained by joining these expressions (or their negations) with the AND logical operator, the integration step took about 1 hour. Most of this time was, indeed, spent for developing simple syntactic transformations to match the input format of the solver. Furthermore, since all the testable entities in this study are expressed in Boolean logic, the SATisPy solver, which we opted to use in the first place due to its ease-of-use, can easily be replaced with any other SAT or CSP solver.

Evaluations. The t-way structure-based U-CIT objects we computed in this study covered all the required t-combinations by construction. Furthermore, the cover-and-generate constructor generally performed better than the generate-and-cover constructor in reducing both the sizes and the construction times (Table 6.7). We, therefore, ran the generate-and-cover constructor with a time-out period of six days per construction. Overall, the cover-and-generate constructor reduced the sizes by an average of 2%, 77%, and 66%, while at the same time reducing the construction times by an average of 3.31%, 95.39%, and 99.56%, when $t = 1$, 2, and 3, respectively. Note further that in 16.67% (6 out of 36) of the experimental setups, te generate-and-cover constructor timed out (Table 6.7). We, therefore, focus on the results obtained from the cover-and-generate constructor in the remainder of this section.

As expected, the higher the coverage strength, the larger the size and the construction time of the structure-based U-CIT objects tended to be. More specifically, the average sizes were 5.50, 36.45, and 195.05 with the average constructions times of 1.87, 40.88, and 1113.18 seconds for 1-, 2-, and 3-way structure-based U-CIT objects, respectively.

Computing t-way DC-adequate covering arrays. Note that as the ultimate goal of the structure-based U-CIT objects is to obtain full coverage under the K_{struct} coverage criterion, they may not cover all the standard t-tuples. For example, the 1-way structure-based U-CIT objects we generated covered 67.33% and 40.00% of all the 2- and 3-tuples, on average, respectively. The numbers were 94.33% and 86.33% for the 2-way structure-based and 95.17% and 91.75% for the 3-way structure-based U-CIT objects.

One good thing about having a seeding mechanism in U-CIT is that it can be

Table 6.8 Information about the t-way DC-adequate covering arrays created by computing 1-way structure-based U-CIT objects using t-way standard covering arrays as seeds. The column '+cfgs.' reports the average numbers of additional configurations needed.

sut	using 2-way standard CAs as seeds				using 3-way standard CAs as seeds			
	generate-and-cover constructor		cover-and-generate constructor		generate-and-cover constructor		cover-and-generate constructor	
	time	+cfgs.	time	+cfgs.	time	+cfgs	time	size
mpsolve	0.72	0.00	0.61	0.00	0.70	0.00	0.60	0.00
dia	0.47	0.00	0.42	0.00	0.45	0.00	0.40	0.00
irissi	1.07	1.00	0.83	1.00	1.05	0.00	0.83	0.00
xterm	0.74	3.80	0.86	1.00	0.77	0.00	0.92	0.00
parrot	3.84	12.40	3.53	7.00	4.25	6.00	4.14	5.00
gimp	5.68	12.20	3.98	3.00	6.28	4.60	4.63	2.00
pidgin	2.53	1.00	3.09	1.00	2.71	0.00	3.31	0.00
python	3.82	5.00	3.51	2.00	3.85	0.00	3.61	0.00
xfig	4.23	3.00	4.24	1.00	4.41	0.00	4.20	0.00
vim	3.72	3.40	4.12	3.00	3.64	0.00	4.14	0.00
sylpheed	5.08	3.40	4.57	2.00	5.71	1.00	5.02	1.00
cherokee	5.80	3.00	6.12	1.00	6.22	1.00	6.13	1.00

Table 6.9 Using structure-based U-CIT objects as seeds to cover the missing 2- and 3-tuples by computing standard covering arrays. The column '+cfgs.' reports the average numbers of additional configurations needed.

sut	standard 2-way CA						standard 3-way CA					
	using t-way structure-based objects as seeds						using t-way structure-based objects as seeds					
	$t=1$		$t=2$		$t=3$		$t=1$		$t=2$		$t=3$	
	time	+cfgs.	time	+cfgs.	time	+cfgs.	time	+cfgs.	time	+cfgs.	time	+cfgs.
mpsolve	0.07	7.00	0.06	2.40	0.06	0.80	0.07	19.40	0.08	13.20	0.07	8.40
dia	0.06	6.80	0.07	2.00	0.07	2.00	0.08	18.80	0.08	11.60	0.08	9.20
irissi	0.07	8.80	0.08	2.00	0.10	2.00	0.16	27.80	0.16	16.60	0.23	34.80
xterm	0.08	9.00	0.09	7.00	0.13	7.00	0.24	31.20	0.24	25.20	0.31	24.40
parrot	0.10	8.40	0.14	4.40	0.24	4.00	0.41	34.40	0.48	21.20	0.66	18.00
gimp	0.15	10.80	0.22	8.20	0.35	7.80	0.97	39.80	1.19	30.40	1.47	28.20
pidgin	0.10	10.00	0.12	7.00	0.21	7.00	0.42	35.40	0.46	24.60	0.69	29.80
python	0.13	10.40	0.19	5.00	0.27	5.00	0.72	38.60	0.70	24.20	1.19	58.60
xfig	0.16	10.80	0.22	3.00	0.34	3.00	1.04	40.00	1.05	21.00	1.43	23.60
vim	0.15	11.00	0.20	3.00	0.34	3.00	0.97	40.20	1.10	20.80	1.40	24.00
sylpheed	0.16	10.60	0.22	4.80	0.39	4.80	1.25	41.00	1.48	24.00	1.65	26.00
cherokee	0.19	12.00	0.27	7.40	0.48	6.80	1.73	43.80	2.24	29.80	2.26	26.00

Table 6.10 Information about the 3-way structure-based U-CIT objects created by using 2-way structure-based U-CIT objects as seeds.

sut	generate-and-cover constructor		cover-and-generate constructor	
	time	size	time	size
mpsolve	45.75	30.00	49.78	34.00
dia	152.62	58.20	65.38	59.60
irissi	2050.65	97.60	1029.66	92.00
xterm	594.90	73.20	167.30	71.60
parrot	18647.29	278.40	4103.58	279.60
gimp	12679.40	216.40	5563.53	215.80
pidgin	52514.79	158.80	30171.88	157.20
python	38510.67	170.40	16897.34	168.20
xfig	59537.75	230.40	14543.20	222.40
vim	67225.58	258.20	19227.10	247.80
sylpheed	117420.77	236.00	67550.43	243.80
cherokee	161779.40	211.20	57712.16	208.40

leveraged to satisfy multiple coverage criteria. For example, one way to obtain t-way DC-adequate covering arrays, i.e., standard t-way covering arrays that guarantee full DC coverage, is to use standard t-way covering arrays as seeds to compute 1-way structure-based U-CIT objects.

To demonstrate the feasibility of this approach, we generated 2- and 3-way DC-adequate covering arrays (Table 6.8). We observed that 1-way structure-based U-CIT objects turned the standard covering arrays into DC-adequate test suites with little increases in both the sizes and the construction times. The average numbers of additional configurations required on top of the standard 2-way and 3-way covering arrays were 1.83 and 0.75, respectively, with the additional construction times of 2.99 and 3.16 seconds, on average.

Note that using structure-based U-CIT objects as seeds to compute standard covering arrays is also possible. To demonstrate the feasibility, we used, 1-, 2-, and 3-way structure-based U-CIT objects as seeds to compute 2- and 3-way standard covering arrays (Table 6.9). The average numbers of additional configurations required on top of the 1-, 2-, and 3-way structure-based U-CIT objects were 21.92, 13.28, and 15.18, respectively, with the additional construction times of 0.40, 0.46, and 0.60 seconds, on average.

Incrementally computing structure-based U-CIT objects. Another use of the seeding mechanism is to leverage lower strength U-CIT objects as seeds for computing higher strength U-CIT objects. To demonstrate the feasibility, we used 2-way structure-based U-CIT objects as seeds to compute 3-way structure-based U-CIT objects. The results of these experiments can be found in Table 6.10.

Computing 4-way structure-based U-CIT objects. Last but not least, we have run

Table 6.11 Information about the 4-way structure-based U-CIT objects created.

sut	valid 4-combs	time	size
mpsolve	1344	16.24	62
dia	32346	111.32	197
xterm	615994	576.92	281
irissi	395504	2067.48	442
pidgin	15293336	16772.38	751
python	19856465	16958.23	869
gimp	14678226	42706.50	1293
parrot	7587625	19631.76	1482
cherokee	47087747	90360.34	2300
sylpheed	81732014	111090.57	3040
xfig	76405845	149335.77	3987
vim	76661558	96900.90	4340

our cover-and-generate constructor for $t = 4$. Table 6.11 presents the results we obtained. Overall, the minimum, the average, and the maximum sizes of the 4-way structure-based U-CIT objects we computed were 62, 1587, and 4340 with the construction times of 16.24, 45544,03, and 96900.90 seconds respectively.

6.1.5 Discussion

Standard covering arrays and structure-based U-CIT objects clearly employ different coverage criteria. We, therefore, do not claim that the U-CIT constructors developed in this work performed better than the standard CIT constructor used in the study. We rather demonstrate that a different CIT constructor is indeed needed to obtain full coverage under the structure-based coverage criterion in an efficient and effective manner. Had the existing constructors addressed the structure-based CIT problem in an efficient and effective manner, there would be no need for U-CIT.

6.2 Order-Based CIT

In this study, we use graphs to model the input spaces of software systems, which we believe can address many interesting test scenarios, such as the ones that arise during the systematic testing of event-driven systems as well as multi-threaded applications. We first define the model of the input space in an abstract manner and briefly discuss

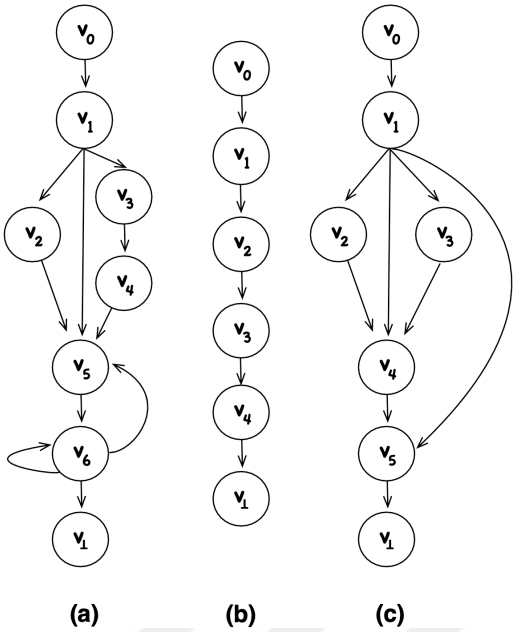


Figure 6.1 Example graph-based models.

two scenarios in which the same or similar models have been used for testing, then present a number of coverage criteria for which CIT can be used to satisfy and discuss the shortcomings of the state-of-the-art CIT approaches, and finally present how U-CIT overcomes these shortcomings.

The model of interest in this study, in its simplest form, is a directed graph $G = (V, E, v_0, v_\perp)$, where V is a set of nodes; E is a set of ordered pairs of the form (v, w), representing a directed edge from node $v \in V$ to node $w \in V$; and $v_0 \in V$ and $v_\perp \in V$ are two distinguished nodes, namely the *entry* and the *exit* node. The entry node has an in-degree of 0 and the exit node has an out-degree of 0. Furthermore, all the nodes are reachable from the entry node and the exit node is reachable from all the nodes. Figure 6.1 presents some example models.

Given a graph-based model, one high-level testing objective is to generate test cases to satisfy some structural coverage criterion, such as exercising every node and/or edge at least once (Samuel & Joseph, 2008). When graphs are used as a model, however, the coverage criterion is often concerned with the order of the entities (e.g., nodes and edges) to be tested. For Figure 6.1a, one such criterion for example, would be to generate a set of paths from the entry node to the exit node, such that every valid order of two (not necessarily distinct) nodes is covered (not necessarily in a consecutive manner) by at least one path. Given this criterion, some example orders to be covered for Figure 6.1a are: $[v_3, v_4]$, $[v_1, v_6]$, $[v_6, v_6]$, and $[v_6, v_5]$, which can all be covered (together with other orders) by the path $(v_0, v_1, v_3, v_4, v_5, v_6, v_5, v_6, v_\perp)$. On the other hand, $[v_2, v_3]$ and $[v_2, v_4]$ are not valid orders since no paths can include

them.

The same and similar graph-based models and coverage criteria have indeed been used for software testing. For example, in systematic testing of event-driven systems, such as graphical user interfaces (GUIs), graph-based models can capture the flow of events in the form of *event sequence graphs* (Belli, Beyazit & Güler, 2011) or *event-flow graphs* (Belli et al., 2011; Memon, 2007), where each node represents an event and a directed edge from v_1 to v_2 indicates that event v_2 can follow event v_1. In this context, an event is considered as an environmental or a user stimulus that from the perspective of testing can be mimicked by a test case. Since the behavior of an event-driven system often depends on the order, in which the events are processed, testing such a system typically involves validating the system response under different event orders.

Another domain, in which graph-based models have been used for testing, concerns the systematic testing of multi-threaded applications. In this domain, the model of a thread captures all sequences of "atomic blocks" that might be traversed through the thread during its execution (Bruening, 1999; Çalpur, 2012). In the remainder of the paper, the aforementioned models are referred to as *atomic block flow graphs* (AFGs).

Each node in an AFG represents an atomic block and the edges connecting the nodes represent the possible execution sequences of atomic blocks. For the programs adhering to a strict mutual-exclusion locking principle (Bruening, 1999), an atomic block is defined as a code segment from one lock exit to the subsequent lock exit. And, a lock exit in this context corresponds to the release of a lock previously acquired for a synchronized code segment (Bruening, 1999; Lu, Jiang & Zhou, 2007; Musuvathi, Qadeer, Ball, Musuvathi, Qadeer & Ball, 2007). For such programs, testing approaches aim to reveal non-deadlock errors, namely *atomicity-violation* and *order-violation* errors. Atomicity-violation errors occur when a sequence of operations that need to be carried in an atomic manner is erroneously divided into multiple atomic blocks, such that the atomicity of the entire operation cannot be guaranteed. Order-violation errors occur when an implicit execution order between two groups of atomic blocks is assumed, but not enforced, e.g., thread A is assumed to start before thread B. To detect these errors, different orders of atomic blocks need to tested.

Note that for these and similar scenarios, since a test case (e.g., a path from the entry node to the exit node) can cover more than one order, the number of test cases to obtain full coverage under a given coverage criterion can be reduced by carefully constructing the test cases. Consequently, CIT approaches can be of practical help.

With standard covering arrays, however, the order of parameter values in a test case is assumed to have no effect on the fault revealing ability of the test case. For example, given a software configuration, such as the ones studied in Chapter 6.1, any permutation of the option settings constituting the configuration, covers exactly the same set of option setting combinations, thus all these permutations should detect the same faulty interactions. For the scenarios we are interested in this study, however, the order matters. Consequently, the types of CIT objects we need in this study are quite different than the one we have computed in Chapter 6.1.

6.2.1 Coverage criterion

To take the orders into account, a different type of covering array, called a *sequence-covering* array, was defined in (Kuhn, Higdon, Lawrence, Kacker & Lei, 2012b) and a number of interesting order-based coverage criteria were presented in (Yuan et al., 2011). In this study, we improve on these works by making both the coverage criteria and the construction approach take the reachability constraints imposed by a given graph-based model into account. Further discussion on this can be found in Chapter 6.2.3.

Definition 16 *Given* $G = (V, E, v_0, v_\perp)$*, a* path *is an ordered sequence of nodes* $(v_{i_1}, \ldots, v_{i_n})$*, such that* $(v_{i_j}, v_{i_{j+1}}) \in E$ *for* $1 \leq j < n$*.*

Definition 17 *Given* $G = (V, E, v_0, v_\perp)$*, a* test case *is a path from* v_0 *to* $v_\perp$*.*

For a given test case p of length n, let p_i, where $0 \leq i \leq n$, be the node located at position i in the test case, such that $p_0 = v_0$ and $p_n = v_\perp$.

Definition 18 *Given* $G = (V, E, v_0, v_\perp)$*, a t-order* $[v_{i_1}, \ldots, v_{i_t}]$*, where* $v_{i_j} \in V$ *for* $1 \leq j \leq t$*, is an ordered tuple of not-necessarily-distinct t nodes, such that there exists a test case p, in which the nodes* $[v_{i_1}, \ldots, v_{i_t}]$ *appear in the order they are given (not necessarily in a consecutive manner, though). A test case p of length n, such that* $p_{m_j} = v_{i_j}$ *and* $0 < m_1 < m_2 < \cdots < m_t < n$ *for* $1 \leq j \leq t$*, is said to cover the t-order* $[v_{i_1}, \ldots, v_{i_t}]$*.*

For instance, for the graph given in Figure 6.1a, $[v_1, v_2]$ and $[v_1, v_6]$, which are both covered by the test case $(v_0, v_1, v_2, v_5, v_6, v_\perp)$, are examples of 2-orders, whereas $[v_2, v_3]$ is not a 2-order, since there is no path from v_2 to v_3.

Definition 19 *Given* $G = (V, E, v_0, v_\perp)$*, a consecutive-t-order* $[v_{i_1}, \ldots, v_{i_t}]$ *is a t-order, such that there exists a test case p of length n, where* $v_{i_j} \in V$ *and* $p_{m_j} = v_{i_j}$

for $1 \leq j \leq t$*, and* $m_{k+1} = m_k + 1$ *for* $1 \leq k < t$*, i.e.,* $[v_{i_1}, \ldots, v_{i_t}]$ *is a subpath in path* p*. Such a test case* p *is said to cover the consecutive-*t*-order* $[v_{i_1}, \ldots, v_{i_t}]$*.*

For instance, for the graph given in Figure 6.1a, $[v_1, v_2]$, $[v_6, v_5]$, and $[v_6, v_6]$, which all appear as subpaths in $(v_0, v_1, v_2, v_5, v_6, v_5, v_6, v_6, v_\perp)$, are examples of consecutive-2-orders, whereas $[v_1, v_6]$, although a 2-order, is not a consecutive-2-order, as there is no edge from v_1 to v_6.

Definition 20 *Given* $G = (V, E, v_0, v_\perp)$*, a non-consecutive-*t*-order* $[v_{i_1}, \ldots, v_{i_t}]$ *is a* t*-order, such that there exists a test case* p *of length* n*, where* $v_{i_j} \in V$ *and* $p_{m_j} = v_{i_j}$ *for* $1 \leq j \leq t$*, and* $m_{k+1} - m_k > 1$ *for at least one* $1 \leq k < t$*. Such a test case* p *is said to cover the non-consecutive-*t*-order* $[v_{i_1}, \ldots, v_{i_t}]$*.*

For Figure 6.1a, $[v_1, v_5]$ is an example of a non-consecutive-2-order, because there is at least one path, e.g., $(v_0, v_1, v_2, v_5, v_6, v_\perp)$, where the nodes constituting the order can appear in a non-consecutive manner. On the other hand, $[v_3, v_4]$, although a 2-order, is not a non-consecutive-2-order, because all the paths including this order have it in a consecutive manner.

Based on these definitions, we define the four coverage criteria given below (inspired from Yuan et al. (2011)). We call this *order-based CIT*.

Definition 21 *Given* $G = (V, E, v_0, v_\perp)$*, a set of test cases* T *is* t-order adequate*, if and only if for every* t*-order in* G*, there exists at least one test case in* T*, which covers it.*

Definition 22 *Given* $G = (V, E, v_0, v_\perp)$*, a set of test cases* T *is* t-cover adequate*, if and only if for every consecutive-*t*-order in* G*, there exists at least one test case in* T*, which covers it.*

Definition 23 *Given* $G = (V, E, v_0, v_\perp)$*, a set of test cases* T *is* t^+-cover adequate*, if and only if for every non-consecutive-*t*-order in* G*, there is at least one test case in* T*, which covers it.*

Definition 24 *Given* $G = (V, E, v_0, v_\perp)$*, a set of test cases* T *is* t^*-cover adequate*, if and only if* T *is both* t*-cover adequate and* t^+*-cover adequate.*

Note that to satisfy the t-order adequacy criterion, all possible t-orders need to be covered at least once regardless of whether they are covered in the form of a consecutive- or non-consecutive-t-order, whereas to satisfy the t-cover adequacy criterion all possible t-orders that can be covered in a consecutive manner need to be covered in the form of a consecutive-t-order. Similarly, to satisfy the t^+-cover adequacy criterion, all possible t-orders that can be covered in a non-consecutive

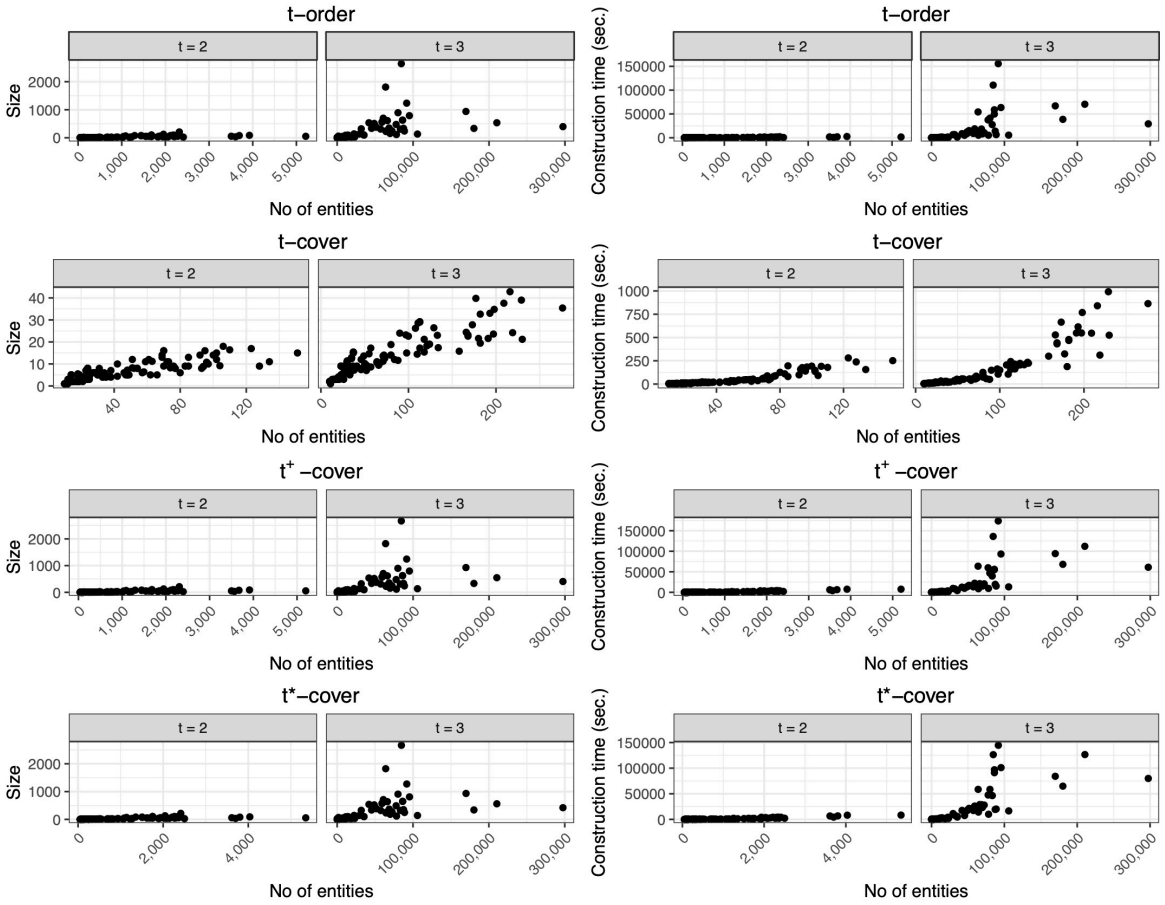


Figure 6.2 Results obtained for the coverage criteria given in Definitions 21-24. The horizontal axes represent the numbers of U-CIT entities to be covered, whereas the vertical axes depict either the average sizes of the U-CIT objects constructed or the average construction times (in seconds), depending on the graph.

manner need to be covered in the form of a non-consecutive-t-order. Finally, t^*-cover adequacy criterion is different than the t-order adequacy criterion, because when a t-order can be covered both in a consecutive and non-consecutive manner, the t^*-cover adequacy criterion guarantees that it is covered in the form of both consecutive- and non-consecutive-t-order, whereas for the t-order adequacy criterion covering it in either way is enough.

6.2.2 Study setup

In this study, we used 171 AFGs obtained from Apache ActiveMQ v5.9.1 (Apache Software Foundation, 2014) – a high-performance, open source message oriented

middleware – to evaluate the proposed approach. We unrolled the cycles in these graphs once to get acyclic graphs, which is a frequently used approach in bounded model checking (Biere, Cimatti, Clarke, Strichman, Zhu & et al., 2003) (see Chapter 6.2.4 for more details). After being unrolled, these graphs had an average of 312.82 nodes ($min = 12$ and $max = 3604$) and an average of 493.23 edges ($min = 12$ and $max = 6566$). All the experiments were carried out on the same Google Cloud platform we used in Study 1 (Chapter 6.1).

Table 6.12 Summary statistics for the construction times (in seconds) and the sizes of the t-way order-based U-CIT objects created for a) $t = 2$ and (b) $t = 3$ b. For each partition, the minimum, median, and maximum values encountered in the partition for the metrics in the columns are reported.

	part.	stat.	nodes	edges	entities	time	size
t-order	1	min	12	12	31	2	1
		med	44	66	61	11	5
		max	46	67	71	18	6
	2	min	14	13	78	3	1
		med	34	38	123	13	3
		max	70	99	224	41	13
	3	min	35	40	227	18	2
		med	59	82	394	38	5
		max	172	248	1020	188	26
	4	min	118	150	1037	116	11
		med	580	871	1936	725	54
		max	3604	6566	5215	2608	210
t-cover	1	min	12	12	10	3	1
		med	28	37	15	5	5
		max	44	66	17	9	5
	2	min	19	23	18	6	2
		med	40	47	18	8	4
		max	54	73	21	11	5
	3	min	23	28	23	7	3
		med	69	98	30	15	5
		max	460	620	52	36	12
	4	min	112	150	54	29	5
		med	580	871	83	92	11
		max	3604	6566	151	399	21
t^+-cover	1	min	12	12	30	12	1
		med	44	66	59	27	5
		max	46	67	68	39	6
	2	min	16	18	73	27	1
		med	34	38	114	50	3
		max	71	111	222	141	13
	3	min	35	40	241	114	2
		med	59	82	397	215	5
		max	172	248	1032	722	47
	4	min	118	150	1089	695	11
		med	580	871	1936	2067	55
		max	3604	6566	5200	7559	213
t^*-cover	1	min	12	12	42	15	1
		med	44	66	76	35	8
		max	46	67	86	53	8
	2	min	16	18	88	35	2
		med	34	38	134	62	4
		max	70	99	240	151	14
	3	min	35	40	242	118	2
		med	59	82	409	232	6
		max	172	248	1065	807	27
	4	min	118	150	1084	644	14
		med	580	871	2029	2234	59
		max	3604	6566	5351	8383	218

(a)

	part.	stat.	nodes	edges	entities	time	size
t-order	1	min	12	12	121	3	1
		med	28	37	312	34	9
		max	44	66	340	49	12
	2	min	17	18	373	5	1
		med	46	67	526	38	8
		max	70	99	1023	81	19
	3	min	34	38	1329	36	3
		med	54	69	3742	90	8
		max	172	248	14056	1612	98
	4	min	99	128	15196	333	24
		med	580	864	65389	12517	347
		max	3604	6566	297308	155572	2643
t-cover	1	min	12	12	10	4	1
		med	28	37	20	11	4
		max	54	69	24	20	9
	2	min	20	24	25	15	3
		med	44	66	28	29	10
		max	50	72	30	33	11
	3	min	23	28	31	17	5
		med	69	98	46	29	10
		max	152	219	73	84	17
	4	min	112	150	80	49	11
		med	580	871	134	234	23
		max	3604	6566	276	1163	42
t^+-cover	1	min	12	12	121	18	1
		med	28	37	310	60	9
		max	44	66	330	82	12
	2	min	17	18	371	39	1
		med	46	67	521	74	9
		max	70	99	1007	156	19
	3	min	34	38	1327	160	3
		med	54	69	3735	277	8
		max	172	248	14034	2865	98
	4	min	99	128	15185	1169	25
		med	580	864	65389	17037	349
		max	3604	6566	297294	173798	2672
t^*-cover	1	min	12	12	136	22	1
		med	28	37	338	79	16
		max	44	66	346	142	18
	2	min	17	18	401	49	3
		med	46	67	544	119	14
		max	70	99	1038	175	26
	3	min	34	38	1373	182	4
		med	54	69	3763	394	15
		max	172	248	14144	3262	102
	4	min	99	128	15253	1359	32
		med	580	864	65507	23023	368
		max	3604	6566	297570	144652	2665

(b)

Note that due to the volume of the data to be reported in this section, using tabular notations was simply out of the question. Therefore, we opted to present different views of the data as we see fit by using plots, such as Figure 6.2, or by using summary tables, such as Tables 6.12-6.13. The raw data can, however, be found at

Table 6.13 Summary statistics for the construction times (in seconds) and the sizes of the 4-way order-based U-CIT objects computed. For each partition, the minimum, median, and maximum values encountered in the partition for the metrics in the columns are reported.

	part.	stat.	nodes	edges	entities	time	size
t-order	1	min	12	12	274	1	1
		med	28	37	1059	34	9
		max	44	66	1432	61	10
	2	min	17	18	1456	3	1
		med	46	67	1735	35	6
		max	51	73	2866	59	15
	3	min	23	28	3189	6	1
		med	44	55	19424	51	6
		max	76	111	27601	786	191
	4	min	41	49	29070	35	4
		med	99	135	118962	559	64
		max	578	804	4360399	92109	5742
t-cover	1	min	12	12	10	2	1
		med	44	55	11	3	3
		max	2889	4956	14	23	7
	2	min	17	18	15	3	2
		med	51	70	18	6	7
		max	1703	3121	21	19	9
	3	min	19	23	23	9	5
		med	56	80	27	16	8
		max	2652	4147	39	67	15
	4	min	59	82	40	19	6
		med	456	702	51	40	14
		max	2748	4264	143	197	26
t^+-cover	1	min	12	12	264	15	1
		med	28	37	1044	81	9
		max	44	66	1426	129	9
	2	min	17	18	1438	43	1
		med	46	67	1724	103	6
		max	51	73	2833	140	15
	3	min	23	28	3174	107	1
		med	44	55	19410	290	6
		max	76	111	27572	1145	191
	4	min	41	49	29046	389	4
		med	112	154	149246	3071	70
		max	578	804	4360336	171050	5821
t^*-cover	1	min	12	12	274	20	1
		med	28	37	1059	93	11
		max	44	66	1432	123	13
	2	min	17	18	1456	47	2
		med	46	67	1735	112	11
		max	51	73	2561	172	17
	3	min	23	28	2866	119	4
		med	40	47	19424	259	7
		max	71	111	27214	1539	199
	4	min	41	49	27601	414	10
		med	99	137	117735	2381	53
		max	578	804	4360399	189773	5858

`https://github.com/susoftgroup/UCIT/`.

In the summary tables, we first divide the experiments into 4 almost equal-size partitions with increasing complexity either by using the number of settings each configuration option has (e.g., Table 6.14) or by using the number of U-CIT entities to be covered (e.g., Tables 6.12 and 6.13). For each partition, we then report the minimum, median, and maximum results obtained in the partition. For a better interpretation of the results, we also filter out the experimental setups, in which the number of testable entities to be covered is less than 10. Furthermore, the partitions are indicated in the summary tables by the unique ID numbers reported under the "part." column.

6.2.3 Applying standard CIT

Modeling. The coverage criteria we have defined in Chapter 6.2.1 are inspired from (Yuan et al., 2011), which empirically demonstrates that these order-based criteria are effective in detecting faults in event-driven software systems, such as graphical user interfaces.

On the other hand, although the aforementioned work presents an approach to generate order-based CIT objects for a given graph-based model, it does not provide a systematic way of taking the reachability constraints imposed by the underlying graph into account during the construction of these objects. Such constraints are rather attempted to be handled after a CIT object is constructed with the aim of converting the invalid test cases, which are erroneously selected due to the overlooked-for constraints, to valid ones. However, no systematic way of carrying this post-mortem analysis is provided in (Yuan et al., 2011). Therefore, this approach can generate many invalid test cases, which may not be trivially "fixed." For example, for the model given in Figure 6.1b, out of 24 possible permutations of 4 nodes (excluding v_0 and $v_\perp$), only one of them (4.2%) is a valid test case, which is difficult to generate by chance. Invalid test cases is an important issue in CIT, because they often result in wasted testing resources (Cohen et al., 2007; Dumlu et al., 2011).

More specifically, the proposed construction approach in (Yuan et al., 2011) uses standard covering arrays to compute order-based CIT objects. It takes as input a set of e events, a coverage strength t, and a predetermined length l for the test cases to be generated (i.e., only fixed-length test cases can be generated) and as

Table 6.14 Summary statistics for the construction times (in seconds) and the sizes of the standard covering arrays obtained by using the order-based construction approach presented in (Yuan et al., 2011). None of the test cases chosen by these standard covering arrays were valid. For each partition, the minimum, median, and maximum values encountered in the partition for the metrics in the columns are reported.

	part.	stats.	options	settings per options	time	size
t=2	1	min	13	9	0.38	175
		med	28	12	0.64	336
		max	29	13	0.67	344
	2	min	15	14	0.47	289
		med	31	17	0.86	561
		max	47	18	2.57	812
	3	min	23	19	0.74	703
		med	39	22	3.22	1118
		max	129	37	80.44	3974
	4	min	54	38	14.99	3474
		med	180	47	377.5	6282
		max	1153	63	43879.56	12171
t=3	1	min	13	9	2.28	2594
		med	29	12	15.07	7014
		max	29	12	19.84	7014
	2	min	15	13	3.8	6496
		med	28	13	19.1	8776
		max	32	15	90.78	14156
	3	min	18	16	3.82	4096
		med	32	17	140.7	20501
		max	47	18	760.67	28240
	4	min	23	19	62.53	24659
		med	33	21	410.34	38834
		max	62	27	11241.67	103923

output computes a standard t-way covering array for l options, each of which can take on e settings (one distinct setting per event). For example, to compute a 2-cover-adequate CIT object for the model given in Figure 6.1a, the aforementioned approach would generate a standard 2-way covering array for 6 options (because the minimum length of a test case to guarantee the coverage of all consecutive-2-orders is 6), each of which has 6 settings (because the number of nodes except for v_0 and $v_\perp$ is 6).

Evaluations. We used the aforementioned construction approach (Yuan et al., 2011) to obtain full coverage under the order-based coverage criteria for the graphs discussed in Chapter 6.2.2. To this end, given a graph with e nodes, we, in an attempt to make sure that every requested order can be covered, used the longest path length l in the graph as the fixed-length. Note that this approach requires us to fix the length of the test cases to be generated. Consequently, the problem of covering different types of t-orders, independent of the actual coverage criterion

used, was turned into a problem of computing a standard t-way covering array for l options, each of which can take on e settings. We used `ACTS` (Yu et al., 2013) to compute the required standard covering arrays. The experiments were repeated 5 times.

Table 6.14 presents the results we obtained. As the graphs got larger, since the number of settings for each option (i.e., the number of nodes e in the graph) increased, it took increasingly longer times to compute the required covering arrays. This was indeed the case even for relatively small option counts (i.e., small values of l). As it was not feasible for us to generate all the required covering arrays, we employed a threshold value of 70 when $t = 2$ and 30 when $t = 3$ on the number of settings an option can have. This enabled us to cover all of the experimental setups, in which $e < 70$ when $t = 2$ and $e < 30$ when $t = 3$. These thresholds were chosen, such that the standard covering array constructor had one day to compute the requested object. Within the allocated time limits, we were able to generate standard covering arrays for 96.49% (165 out of 171) of the models when $t = 2$ and for 66.67% (114 out of 171) of the models when $t = 3$.

None of the test cases in the generated covering arrays, on the other hand, were valid. As no systematic approach is presented in (Yuan et al., 2011) to take the reachability constraints enforced by the underlying graphs into account or to fix the invalid test cases in a post-mortem manner, it is was not clear at all how to avoid and/or fix these invalid test cases.

6.2.4 Applying U-CIT

Given a graph $G = (V, E, v_0, v_\perp)$, which models the input space of the system under test, an U-CIT testable entity corresponds to a t-order, a consecutive-t-order, or a non-consecutive-t-order to be covered, depending on the the coverage criterion (Definitions 21-24). Then, an U-CIT test case corresponds to a path from v_0 to $v_\perp$ (Definition 17). Finally, the graph G, as it restricts the orders to be covered as well as the test cases to be generated, is expressed as the U-CIT model constraint C.

To this end, we encode the problem of finding a path from a source node to a sink node as a single-source single-sink flow problem (Ahlswede, Cai, Li & Yeung, 2000). In particular, flow on an edge $(v_i, v_j) \in E$ (using the terminology of flow networks) is represented by a unique variable e_{ij}. From the perspective of finding a path, an edge $(v_i, v_j) \in E$ is either taken, i.e., $e_{ij} = 1$, indicating that there is a flow on the

edge, or not taken, i.e., $e_{ij} = 0$, indicating that there is no flow on the edge:

$$e_{ij} \in \{0, 1\}. \tag{6.1}$$

Leaving cyclic graphs aside for the moment (which will be discussed later on in this section), to generate a test case, i.e., to form a flow from the source node v_0 to the sink node $v_\perp$, one of the outgoing edges of v_0 and one of the incoming edges of $v_\perp$ must be taken:

$$\sum_{(v_0, v_i) \in E} e_{0i} = 1 \tag{6.2}$$

$$\sum_{(v_i, v_\perp) \in E} e_{i\perp} = 1. \tag{6.3}$$

Note that since the graph is acyclic, at most one of the incoming and at most one of the outgoing edges of a node can be taken, i.e., there can be a flow on at most one incoming and at most one outgoing edge.

The flow through node i is then expressed as a constraint indicating that the amount of outgoing flow from i is the same as the amount of incoming flow to i:

$$\sum_{(v_k, v_i) \in E} e_{ki} = \sum_{(v_i, v_l) \in E} e_{il} \ \leq \ 1. \tag{6.4}$$

Note that the source and the sink nodes are exempt from (6.4) since there is no flow into the source node and no flow out of the sink node.

As an example, Figure 6.3 presents an encoding to compute a test case, i.e., a path from v_0 to $v_\perp$, for the graph given in Figure 6.1c.

To make sure that a specific order is covered by a test case, additional constraints are needed. More formally, to cover a t-order $[v_{i_1}, \ldots, v_{i_t}]$ in a graph $G = (V, E, v_0, v_\perp)$, the following additional constraints are needed:

$$e_{01}, e_{12}, e_{13}, e_{14}, e_{15}, e_{24}, e_{34}, e_{45}, e_{5\perp} \in \{0,1\}$$
$$e_{01} = 1$$
$$e_{5\perp} = 1$$
$$e_{01} = e_{12} + e_{13} + e_{14} + e_{15}$$
$$e_{12} = e_{24}$$
$$e_{13} = e_{34}$$
$$e_{24} + e_{14} + e_{34} = e_{45}$$
$$e_{45} + e_{15} = e_{5\perp}$$

Figure 6.3 Single-source single-sink encoding to find a path from the entry node v_0 to the exit node $v_\perp$ in the graph given in Figure 6.1c.

$$\sum_{(v_k, v_{i_s}) \in E} e_{ki_s} = 1 \; for \; 1 \leq s \leq t, \tag{6.5}$$

which indicate that all the nodes in the requested order must be visited. Since the graph is acyclic, the order of visit is guaranteed.

For example, to cover the 3-order $[v_1, v_4, v_5]$ in Figure 6.1c, the encoding in Figure 6.3 needs to be extended with:

$$\begin{aligned} e_{01} &= 1 \\ e_{24} + e_{14} + e_{34} &= 1 \\ e_{45} + e_{15} &= 1. \end{aligned} \tag{6.6}$$

To cover the same order $[v_{i_1}, \ldots, v_{i_t}]$ in a non-consecutive manner, however, the following constraint is required in addition to (6.5):

$$\sum_{1 < s \leq t} e_{i_{s-1} i_s} < t - 1, \tag{6.7}$$

which ensures that the length of the path from v_{i_1} to v_{i_t} is at least t.

Going back to our running example, to cover $[v_1, v_4, v_5]$ in a non-consecutive manner,

the following additional constraint is required on top of (6.6):

$$e_{14} + e_{45} < 2. \tag{6.8}$$

If, on the other hand, the t-order $[v_{i_1}, \ldots, v_{i_t}]$ needs to be covered in a consecutive manner, then the following constraints are needed instead of (6.5) and (6.7):

$$e_{i_{s-1}i_s} = 1 \; for \; 1 < s \leq t, \tag{6.9}$$

making sure that the edges between all the consecutive pairs of nodes in the order are taken.

For our running example in Figure 6.3, we would need the following additional constraints to cover $[v_1, v_4, v_5]$ in a consecutive manner:

$$\begin{aligned} e_{14} &= 1 \\ e_{45} &= 1. \end{aligned} \tag{6.10}$$

For a given graph $G = (V, E, v_0, v_\perp)$, we have, therefore, defined the U-CIT model in this study as $M =< P, D, C >$, where P is the set of variables, each of which represents a distinct edge in the graph; D is a set of sets $\{0, 1\}$, one per edge, indicating whether there is flow on the edge or not (i.e., whether the edge is taken or not); and C is the model constraint capturing the reachability restrictions in the graph. More specifically, for a given graph, Equations 6.1-6.4 constitute the model constraint C. For example, Figure 6.3 presents the U-CIT model constraint created for the graph given in Figure 6.1c. Note that, given a graph, C stays the same regardless of the testable entities to be covered. Each U-CIT testable entity then corresponds to an order to be covered (Definitions 21-24). In particular, to cover a t-order, Equation 6.5; to cover a t-order in a non-consecutive manner, Equations 6.5 and 6.7; and to cover a t-order in a consecutive manner, Equation 6.9 needs to be used. As an example, Equation 6.6 presents the constraints to be used to cover the 3-order $[v_1, v_4, v_5]$ in the graph given in Figure 6.1c. Similarly, Equations 6.6 and 6.8 are needed to cover the same 3-order in a non-consecutive manner. And, Equation 6.10 is needed to cover it in a consecutive manner. Each U-CIT test case then corresponds to a path from the entry node v_0 to the exit node $v_\perp$ (Definition 17),

```
% graph
edge(v0, v1).
edge(v1, v2).

% 'reaches' definitions
reaches(A, B) :- edge(A, B).
reaches(A, B) :- edge(A, C), reaches(C, B).

% 3-orders
order(A, B, C) :- reaches(A, B), reaches(B, C).

% consecutive-3-orders
consec(A, B, C) :- edge(A, B), edge(B, C).

% non-consecutive-3-orders
nonconsec(A, B, C) :- reaches(A, X), X!=A, X!=B, reaches(X, B), reaches(B, C).
nonconsec(A, B, C) :- reaches(A, B), reaches(B, X), X!=B, X!=C, reaches(X, C).
```

Figure 6.4 An example ASP encoding for determining the valid consecutive, nonconsecutive, and regular 3-orders.

covering a number of required orders.

Note that we have so far been concerned with directed acyclic graphs (DAGs). To work with cyclic graphs, we unroll the cycles k times (for this work, $k = 1$), which is a frequently used approach in bounded model checking (Biere et al., 2003). To this end, we first convert a given graph to a regular expression (Hopcroft, 2013), where all the Kleene plus operators are replaced by using the Kleene star operator, i.e., converting a^+ to aa^*. We then replace all the Kleene stars in the expression using the bounded repetition operator, such that the respective strings can be repeated at most k times, i.e., converting a^* to $a\{0, k\}$. Finally, the resulting regular expression is converted back to a graph.

For this work, we used the `Vcsn` tool (Lombardy, Régis-Gianas & Sakarovitch, 2004) to carry out these steps. More specifically, converting a graph to a regular expression and a regular expression to a graph were carried out by using a single `Vcsn` shell command. And, replacing the unbounded Kleene star operators by bounded repetition operators was performed by another shell command using the `replace` string-replacement utility.

After having an acyclic graph, we used ASP (Answer Set Programming) (Marek & Truszczyński, 1999; Niemelä, 1999) to determine the different types of t-orders to be covered. Note that this step could also have been carried out by using reachability-based graph algorithms. We, however, chose to use ASP because, being a declarative logic programming paradigm, it was a perfect match for the task at hand. We were even able to provide whole code segments in the paper (e.g., Figure 6.4) to demonstrate the effort involved in the development

Figure 6.4 presents an example ASP encoding to determine the consecutive, non-consecutive, and regular 3-orders. Below, we explain the encoding in a nutshell with no intention to introduce ASP. For more details about ASP, the interested reader may refer to an introduction (Eiter, Ianni & Krennwallner, 2009) or a book (Baral, 2003).

A DAG is expressed by using `edge(..)` facts. There is a path from node `A` to node `B`, i.e., `A` reaches `B` or `B` is reachable from `A`, if there is an edge from `A` to `B` (i.e., `edge(A, B)` holds) or there is an edge from `A` to `C` and `B` is reachable from `C`:

```
reaches(A, B) :- edge(A, B).

reaches(A, B) :- edge(A, C), reaches(C, B).
```

Then, `[A, B, C]` is a valid 3-order, i.e., `order(A, B, C)`, if `A` reaches `B` and `B` reaches `C`:

```
order(A, B, C) :- reaches(A, B), reaches(B, C).
```

For a 3-order `[A, B, C]` to be covered in a consecutive manner, there should be an edge from `A` to `B` and edge from `B` to `C`:

```
consec(A, B, C) :- edge(A, B), edge(B, C).
```

And, for the same order to be covered in a non-consecutive manner, `B` should be reachable from `A` via another node or `C` should be reachable from `B` via another node:

```
nonconsec(A, B, C) :- reaches(A, X), X!=B, reaches(X, B),
                                           reaches(B, C).

nonconsec(A, B, C) :- reaches(A, B), reaches(B, X), X!=C,
                                           reaches(X, C).
```

Note that this encoding can trivially be extended to determine t-orders for any strength t.

Cost. All told, developing a generic Python script to unroll the cycles in a given graph using `Vcsn` took about 2 hours, which was mostly spent for writing procedures to match the input and output formats of `Vcsn`. Similarly, developing a generic Python script to automatically generate the ASP encodings for determining different types of t-orders to be covered, such as the one given in Figure 6.4, took about another 2 hours. Integrating a CSP solver, namely Sugar (Tamura & Banbara, 2008), with the constructors (as also discussed in Chapter 6.3.4) took less than 1 hour.

Evaluations. To evaluate the proposed approach, we first used our U-CIT constructors to compute t-way ($t = \{2,3\}$) order-based U-CIT objects for the graphs discussed in Chapter 6.2.2. The experiments were repeated up to 5 times; 94% of the experiments with the best-performing U-CIT constructor (i.e., cover-and-generate) were repeated exactly 5 times. By construction, all the test cases selected by the U-CIT objects computed were valid and all these U-CIT objects achieved full coverage under the respective coverage criterion.

As was the case with the previous study (Chapter 6.1), the cover-and-generate constructor performed generally better than the generate-and-cover constructor. Therefore, we ran the generate-and-cover constructor with a time-out period of one day, while letting the cover-and-generate constructor run to completion. For 93.20% (1275 out of 1368) of the experimental setups, the generate-and-cover constructor computed the requested U-CIT objects within the allocated time limits. For these setups, the cover-and-generate constructor reduced the sizes by an average of 65.86% and 60.79%, while at the same time reducing the construction times by an average of 72.24% and 77.03% when $t = 2$ and 3, respectively. We, therefore, focus on the results obtained from the cover-and-generate constructor in the remainder of this section.

Figure 6.2 presents the results obtained from the cover-and-generate constructor and Table 6.12 provides some summary statistics. As expected, the coverage criteria listed in the order of increasing number of entities they required to cover, were: t-cover, t^+-cover, t-order, and t^*-cover. These criteria respectively marked an average of 38.50, 665.87, 672.37, and 704.36 entities for coverage when $t = 2$; and an average of 61.73, 21678.98, 21685.99, and 21740.71 entities when $t = 3$.

The sizes of the order-based U-CIT objects as well their construction times tended to be correlated with the number of entities to be covered. Overall, the minimum, the average, and the maximum sizes of the U-CIT objects created were 1, 17.28, and 220, respectively, when $t = 2$; and 1, 54.27, and 2672 when $t = 3$. And the construction times for these objects respectively were 2.57, 461.32, and 4156.25 seconds when $t = 2$; and 3.24, 2568.12, and 136116.19 seconds when $t = 3$.

Another trend we observed was that although the numbers of entities to be covered by the t-order criterion were similar to those to be covered by the t^+-cover and t^*-cover criteria, covering the latter set of entities took longer than covering the former set of entities. The average constructions times were 251.29, 751.03, and 806.92 seconds for t-order, t^+-cover, and t^*-cover criteria, respectively, when $t = 2$; and 6885.59, 9508.46, and 10626.21 seconds when $t = 3$. We believe that this was because of the additional constraints to be satisfied to make sure that the requested

orders are covered in a non-consecutive manner (i.e., need for solving the constraints in Equation 6.7 on top of Equation 6.5).

Computing 4-way order-based U-CIT objects. Last but not least, we ran our cover-and-generate constructor for $t = 4$ with a time-out period of 200 hours. For 88.01% (602 out of 684) of the experimental setups, the constructor was able to generate the requested U-CIT objects within the allocated time limits, whereas for the remaining 11.99% of the setups, it timed out.

Table 6.13 presents the results we obtained. Overall, the minimum, the average, and the maximum sizes of the 4-way order-based U-CIT objects computed were 1, 136.42, and 5858, respectively. And the construction times for these objects respectively were 1.87, 4522.53, and 189773.20 seconds.

6.2.5 Discussion

Note that given a graph, there are different approaches for solving the problem of finding a path covering certain sequences of nodes. In this study, however, our goal was to demonstrate that there is at least one solution, which can be expressed in U-CIT. For example, instead of using a constraint solver, one can use a model checker and formulate the same problem as a property stating that there is no path covering the requested orders. A counter example (if any) would then be a test case covering the orders. Similarly, one can even develop a special purpose constraint solver, which uses graph-based reachability algorithms, to determine whether a given set of orders can appear on a single path. These solutions would all work with U-CIT as long as the underlying solver supports the single primitive *solve* as discussed in Chapter 5.

6.3 Usage-Based CIT

An electronics company has approached us to improve their CIT-based testing practices. In particular, they were interested in testing the Internet connectivity feature of a consumer device, which they market in dozens of countries. The end-users of this device can customize the aforementioned feature by using 9 configuration options, which have 308, 280, 154, 82, 58, 41, 6, 3, and 2 settings, respectively. Since

there is no system-wide constraint, all possible configurations (i.e., all possible combinations of option settings) are valid. All told, these options constitute a space of more than 90 billion valid configurations.

The company provided us with 526691 real configurations that they collected from the field during the month of May in 2016. Each configuration was obtained from a different consumer device and there were a total of 37503 distinct configurations, i.e., some configurations were used by multiple costumers.

Historically, configuration-related failures in this system have often been caused by the faulty interactions among the configuration options. However, exhaustive testing of neither the whole configuration space nor the distinct configurations seen in the field, is desirable for the company. Due to legal and privacy concerns, we are not able to provide further details.

6.3.1 Coverage criterion

We first attempted to create standard covering arrays for the scenario at hand (see Chapter 6.3.3 for more information). It turned out that the smallest covering array we could generate was a 2-way covering array of size 86241. It is, however, quite difficult to justify the use of all these configurations for testing when one knows that the total number of distinct configurations used in the field is 37503. Had the company had enough resources (i.e., time and computing platforms) to test all the distinct configurations in the field, they would have done it.

We, therefore, defined two novel coverage criteria, namely K_{seen} and $K_{weighted}$, based on the idea that when testing all t-tuples is not feasible, one should at the very least, consider testing the t-tuples appearing in the field. We call this *usage-based CIT*.

Definition 25 *The* seen-t-way coverage criterion K_{seen} *takes as input a set of configurations T, a coverage strength t, and a* cutoff frequency *in* $[0,1)$*, and mark for coverage all the t'-tuples ($1 \leq t' \leq t$) appearing in T, the frequencies of which are greater than the cutoff frequency.*

The frequency of a tuple is computed as follows:

Definition 26 *Given a set of configurations T, the* frequency *of a tuple is the ratio of the number of configurations in T, in which the tuple appear, to the total number of configurations in T.*

Note that, when the frequency cutoff is 0, K_{seen} selects all the t'-tuples ($1 \leq t' \leq t$) appearing in T.

K_{seen} can further be extended to obtain variable strength coverage by using a weighted sum of the frequencies, where the weight of a tuple is defined as follows:

Definition 27 *Given a set of configurations T, the* weight *of a tuple is the ratio of the number of times the tuple appears in T to the total number of tuples in T.*

Note that computing the denominator in Definition 27 does not require to explicitly enumerate all possible tuples appearing in T. More specifically, since the number of tuples in a given configuration of k options is $2^k - 1$, the total number of tuples in T (thus the denominator) is $|T|(2^k - 1)$.

Definition 28 *The* weighted-t-way coverage criterion $K_{weighted}$ *takes as input a set of configurations T, a coverage strength t, and a* cutoff weight *in* $(0, 1]$*, and mark for coverage a minimal set of t'-tuples ($1 \leq t' \leq t$), the total weight of which is greater than or equal to the given cutoff weight.*

To determine the tuples to be covered by this criterion, all the t'-tuples ($1 \leq t' \leq t$) appearing in T are sorted by the descending order of their weights. Then, the minimum number tuples from the top of the list are selected, such that the total weight of the selected tuples is greater than or equal to the cutoff weight. Note that the $K_{weighted}$ criterion with the cutoff weight of 1 can be satisfied by selecting all the distinct configurations in T.

6.3.2 Study setup

For the evaluations, we used the aforementioned subject application with 9 configuration options, which had 308, 280, 154, 82, 58, 41, 6, 3, and 2 settings, respectively, together with the 526691 real configurations collected from the field, out of which 37503 were distinct.

All the experiments were carried out on the same Google Cloud platform with the previous two studies (Chapters 6.1 and 6.2).

6.3.3 Applying standard CIT

Modeling. We first attempted to create standard covering arrays of various strengths by using a number of well-known covering array constructors, namely Jenny (Jenkins, 2005), PICT (Czerwonka, 2008), and ACTS (Yu et al., 2013).

The very first thing we observed was that although we had a small number of configuration options (only 9), due to the large number of settings some of these options had, many of the existing covering array constructors failed to generate the requested covering arrays. For example, we were not even able to model the configuration space in Jenny, because it turned out Jenny employs the letters of the English alphabet to represent the settings of a configuration option, limiting the maximum number of settings that an option can have to 52 (the number of capital and lowercase letters in the English alphabet). On the other hand, PICT, which is specifically designed for scalability (Czerwonka, 2008), was able to generate a 2-way covering array of size 86241 in 100 seconds. It, however, failed to generate a 3-way covering array in 10 days, after which we terminated the process. Whereas ACTS was able to generate a 2-way covering array of size 86255 in 16 seconds and a 3-way covering array of size 13283730 in 1887 seconds (about 32 minutes). However, when we attempted to generate 4-way covering arrays, ACTS crashed after a while with some memory-related errors.

Note that given a usage-based coverage criterion, neither the tuples to be covered nor the tuples not to be covered can be expressed as constraints in standard constructors in an attempt to selectively determine what to cover and what not to cover. This is because constraints in standard constructors are globally enforced, i.e., all of the test cases selected must satisfy all of the constraints. Therefore, expressing a tuple, which is selected by a given coverage criterion, as a constraint to indicate that the tuple needs to be covered, will enforce the same tuple to appear in all of the selected configurations. Since this can prevent conflicting tuples from being covered, no covering array can be created. Similarly, expressing a tuple, which is not selected by a given coverage criterion, as a constraint to indicate that the tuple needs to be avoided, can also prevent the creation of a covering array. It may not, for example, be possible to assign values to certain model parameters due to some invalidated tuple combinations.

An alternative approach might be to express tuples that are not needed to be covered as *soft constraints*, which mark combinations of parameter values that are permitted, but not desirable (Bryce & Colbourn, 2006). However, when the tuples to be covered is a small fraction of all the tuples, the number of soft constraints can get quite large, which can in turn cause performance and scalability issues. For example, in our experiments, 99.90% of all the tuples (of strength up to and including a given value

Table 6.15 Statistics about the K_{seen} coverage obtained by standard covering arrays. The columns, respectively, report the frequency cutoff values, the numbers of testable entities to be covered, and the numbers of testable entities covered by the standard 2-way and 3-way covering arrays created for the study.

	t=2			t=3			t=4			t=5			t=6		
		% covered			% covered			% covered			% covered			% covered	
	no of	by standard CAs		no of	by standard CAs		no of	by standard CAs		no of	by standard CAs		no of	by standard CAs	
cutoff	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way
0.5	4	100	100	4	100	100	4	100	100	4	100	100	4	100	100
0.25	20	100	100	25	100	100	26	100	100	26	100	100	26	100	100
0.2	34	100	100	44	100	100	49	100	100	50	98	100	50	98	100
0.15	54	100	100	89	97	100	100	94	100	101	93	100	101	93	100
0.1	80	100	100	164	90	100	235	78	100	264	70	97	269	69	96
0.05	200	100	100	474	85	100	734	67	96	900	56	87	971	51	81
0.04	240	100	100	601	84	100	964	65	95	1204	53	85	1318	48	79
0.03	299	100	100	811	80	100	1395	58	94	1811	46	82	2001	42	75
0.02	422	100	100	1281	76	100	2382	53	92	3256	39	79	3705	35	71
0.01	669	100	100	2201	74	100	4475	47	90	6585	33	74	7825	28	63
0.005	1056	100	100	3751	71	100	7949	44	89	12028	30	71	14634	25	59
0.001	2652	100	100	11604	67	100	27890	39	85	45599	25	65	57741	19	53
0	22554	100	100	182952	42	100	658825	11	67	1240182	1	27	1321685	0	4

Table 6.16 Statistics about the $K_{weighted}$ coverage obtained by standard covering arrays. The columns, respectively, report the weight cutoff values, the numbers of testable entities to be covered, and the numbers of testable entities covered by the standard 2-way and 3-way covering arrays created for the study.

	t=2			t=3			t=4			t=5			t=6		
		% covered			% covered			% covered			% covered			% covered	
	no of	by standard CAs		no of	by standard CAs		no of	by standard CAs		no of	by standard CAs		no of	by standard CAs	
cutoff	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way	entities	2-way	3-way
0.70	332	100	100	2171	74	100	8037	44	89	18518	28	69	29272	22	56
0.75	426	100	100	2951	72	100	11240	42	87	26467	27	68	42694	21	54
0.80	566	100	100	4117	70	100	16252	41	87	39309	25	66	63498	19	52
0.85	793	100	100	6051	69	100	24798	39	86	60222	23	64	98886	17	50
0.90	1185	100	100	9675	68	100	40330	37	85	101015	21	62	169254	15	47
0.95	2073	100	100	17826	64	100	79330	33	83	211007	17	58	368650	12	43

of t), on average, did not need to be covered. In other words, had soft constraints been used to express these need-not-to-be-covered tuples, the number of constraints would have been as high as 4.7 trillion in some experiments. We couldn't experiment with this approach, because none of the standard constructors that we have access to, supported soft constraints.

Evaluations. All told, the size of the smallest standard covering array that we could generate was larger than the number of distinct configurations seen in the field, which rendered the use of standard covering arrays in this context hard to justify.

To further demonstrate that obtaining full coverage in an efficient and effective manner under the usage-based coverage criteria is a non-trivial task, Tables 6.15-6.16 report the coverage percentages obtained by the standard covering arrays generated in this study. In particular, when $t > 3$ with K_{seen}, the standard 2- and 3-way covering arrays did not guarantee to cover all the requested tuples. For example, when cutoff=0.001, only 39% (85%), 25% (65%), and 19% (53%) of all the required tuples for $t = 4$, 5, and 6 under K_{seen}, were covered by the standard 2-way (3-way) covering arrays (Table 6.15). Similarly, the standard covering arrays did not

guarantee to cover all the tuples requested by $K_{weighted}$ either, especially for large values of coverage strength and weight cutoff values. For example, when $t = 6$ and cutoff=0.95 only 12% and 43% of all the required tuples were covered by the standard 2-way and 3-way covering arrays, respectively (Table 6.16).

6.3.4 Applying U-CIT

Modeling. We have defined the U-CIT model as $M =< P, D, C >$, where $P = \{o_1, \ldots, o_9\}$, $D = \{\{1..308\}, \{1..280\}, \{1..154\}, \{1..82\}, \{1..58\}, \{1..41\}, \{1..6\}, \{1..3\}, \{1, 2\}\}$, and $C : true$, indicating that all possible configurations were valid.

Each U-CIT testable entity then naturally corresponded to a tuple selected by the coverage criterion K_{seen} or $K_{weighted}$, which was expressed as a constraint over finite sets. For example, the 3-tuple $(o_1 = 204, o_5 = 12, o_9 = 1)$ was expressed as $o_1 = 204 \wedge o_5 = 12 \wedge o_9 = 1$. Note that the very same approach can readily be used to define and compute standard t-way covering arrays as U-CIT objects by expressing all valid t-tuples as U-CIT testable entities.

Consequently, any solver that works with logical operators, such as $\wedge$ (AND), and equality constraints over finite sets, such as $o_1 = 204$, including the commonplace SAT and CSP solvers (de Moura & Bjørner, 2009; Katebi, Sakallah & Marques-Silva, 2011), can be used with the U-CIT constructors compute the U-CIT objects satisfying the K_{seen} and $K_{weighted}$ criteria.

Indeed, being able to work with any type of constraints as long as an appropriate solver is provided, improves the flexibility of U-CIT. To demonstrate that this feature also enables the use of domain- and/or application-specific solvers, we have implemented a quite simple solver for this study, instead of trivially using an existing SAT or CSP solver.

Algorithm 3 presents the aforementioned solver. It simply determines whether a given set of tuples E can be accommodated together in a single configuration or not. In particular, it marks E as satisfiable as long as the option settings appearing in E do not contradict with each other (lines 7-8).

Cost. All told, developing a generic script to determine the tuples (i.e., the testable entities) selected by the K_{seen} and $K_{weighted}$ coverage criteria for any configuration space model, coverage strength, and cutoff value, took less than 2 hours. And, im-

Algorithm 3 Determine if a given set of tuples can be accommodated together in a configuration cfg.

Input: Set of tuples E
Output: $True$ or $False$

```
1: cfg ← undef
2: for each tuple e in E do
3:     for each option o in e do
4:         Let e[o] is the value of o in e
5:         Let cfg[o] is the value of o in cfg, which is initially undef
6:         if defined cfg[o] and cfg[o] ≠ e[o] then
7:             return False
8:         else
9:             cfg[o] = e[o]
10:        end if
11:    end for
12: end for
13: return True
```

plementing the solver in Algorithm 3 and integrating it with the U-CIT constructors took less than 1 hour. To further demonstrate the flexibility of U-CIT, we have also integrated our constructors with a CSP solver (namely, Sugar (Tamura & Banbara, 2008)) to solve exactly the same set of constraints. Interestingly enough, it took about the same time (less than 1 hour) for us to do that as we needed to implement a simple procedure to match the input format of the solver. The implementation was done in Python.

Evaluations. To evaluate the proposed approach, we carried out a series of experiments. In these experiments, we used the cover-and-generate and generate-and-cover constructors given in Algorithms 1 and 2 to compute U-CIT objects of various strengths. Since the cover-and-generate constructor performed generally better than the generate-and-cover constructor, the experiments with the latter constructor were repeated up to 3 times and with a time-out period of one day for each repetition to keep the cost of the experiments under control. The experiments with the former constructor, on the other hand, were repeated 100 times to evaluate the sensitivity of the proposed approach to the order, in which the testable entities are processed, except for the experimental setups where the frequency cutoff was 0 and $t > 2$, which were repeated only once, to keep the cost under further control. In all the experiments, the orders were randomly generated by shuffling the testable entities to be covered.

Evaluating the K_{seen} coverage criterion. Table 6.17 summarizes the results we obtained for the K_{seen} coverage criterion. We first observed that the U-CIT con-

Table 6.17 Statistics about the U-CIT objects created for the K_{seen} coverage criterion, where the columns, respectively, report the coverage strengths, the frequency cutoff values, the numbers of testable entities to be covered, and the average construction times (in seconds) as well as the average sizes of the U-CIT objects computed by the generate-and-cover and cover-and-generate constructors together with the minimum, maximum, standard deviation, and coefficient of variation statistics for the results obtained from the latter constructor. The character '*' marks the experimental setups, in which the generate-and-cover constructor timed out after one day. Furthermore, the number of times the experiments were repeated are given in the column "repeat count."

			generate-and-cover constructor			cover-and-generate constructor									
						time					size				
t	cutoff	no of entities	avg. time	avg. size	repeat count	min.	avg.	max.	sd.	cv.	min.	avg.	max.	sd.	cv.	repeat count
2	0.5	4	0.02	2.00	3	0.00	0.03	0.06	0.01	57.86	1	1.00	1	0.00	0.00	100
2	0.25	20	0.05	8.67	3	0.01	0.08	0.19	0.05	56.35	2	2.31	3	0.46	20.02	100
2	0.2	34	0.13	16.00	3	0.01	0.13	0.27	0.07	56.09	3	3.89	5	0.61	15.80	100
2	0.15	54	0.25	24.00	3	0.00	0.18	0.38	0.11	61.40	4	4.70	7	0.62	13.29	100
2	0.1	80	0.60	38.33	3	0.02	0.29	0.60	0.16	56.09	5	6.50	9	0.83	12.78	100
2	0.05	200	3.02	94.33	3	0.06	0.73	1.54	0.41	55.85	13	16.00	19	1.33	8.34	100
2	0.04	240	4.51	124.00	3	0.06	0.93	1.95	0.52	55.76	19	21.72	25	1.46	6.74	100
2	0.03	299	6.05	152.33	3	0.04	1.14	2.40	0.66	58.07	21	27.46	31	1.77	6.46	100
2	0.02	422	10.47	211.33	3	0.14	1.74	3.61	0.97	55.53	35	40.53	45	1.93	4.75	100
2	0.01	669	23.79	333.00	3	0.32	3.07	6.07	1.67	54.36	61	69.34	75	2.82	4.06	100
2	0.005	1056	50.97	518.33	3	0.30	4.92	9.60	2.75	55.91	113	118.83	126	2.91	2.44	100
2	0.001	2652	265.32	1275.00	3	2.29	14.85	27.08	7.51	50.58	311	327.31	342	5.89	1.80	100
2	0	22554	10389.32	9606.33	3	228.72	364.05	493.08	75.38	20.71	3589	3625.24	3666	17.06	0.47	100
3	0.5	4	0.01	2.33	3	0.00	0.02	0.05	0.01	56.45	1	1.00	1	0.00	0.00	100
3	0.25	25	0.10	8.33	3	0.01	0.08	0.18	0.04	58.47	2	2.26	3	0.44	19.41	100
3	0.2	44	0.12	15.33	3	0.01	0.12	0.29	0.07	58.07	3	3.82	6	0.77	20.07	100
3	0.15	89	0.61	32.00	3	0.01	0.16	0.38	0.10	58.08	4	4.96	8	0.95	19.11	100
3	0.1	164	1.20	53.67	3	0.03	0.28	0.68	0.17	57.97	5	6.86	10	1.33	19.34	100
3	0.05	474	8.16	170.00	3	0.06	0.79	2.66	0.50	63.02	14	18.54	24	2.22	12.00	100
3	0.04	601	10.22	194.00	3	0.13	1.08	2.09	0.57	52.87	20	26.24	32	2.32	8.84	100
3	0.03	811	20.99	285.33	3	0.12	1.37	2.82	0.74	54.17	27	34.60	42	3.14	9.08	100
3	0.02	1281	40.73	456.00	3	0.19	2.30	4.48	1.26	54.66	45	57.47	68	3.95	6.87	100
3	0.01	2201	106.93	781.67	3	0.58	4.66	9.26	2.48	53.10	103	113.27	129	4.67	4.13	100
3	0.005	3751	251.85	1340.67	3	1.26	9.05	17.85	4.59	50.71	203	217.86	238	6.87	3.15	100
3	0.001	11604	1813.94	4137.00	3	14.21	43.31	71.85	16.66	38.46	790	817.66	853	12.96	1.59	100
3	0	182952	197208.59	72642.00	2	8266.04	8266.04	8266.04	n/a	n/a	24971	24971.00	24971	n/a	n/a	1
4	0.5	4	0.02	2.33	3	0.00	0.02	0.06	0.01	60.87	1	1.00	1	0.00	0.00	100
4	0.25	26	0.05	7.67	3	0.01	0.08	0.18	0.04	58.93	2	2.31	3	0.46	20.02	100
4	0.2	49	0.15	16.00	3	0.01	0.11	0.28	0.07	59.04	3	3.65	5	0.73	19.90	100
4	0.15	100	0.43	24.33	3	0.01	0.15	0.37	0.09	57.23	4	4.91	8	0.91	18.46	100
4	0.1	235	1.69	65.33	3	0.01	0.24	0.55	0.14	58.37	5	6.32	10	1.22	19.23	100
4	0.05	734	10.29	188.33	3	0.07	0.66	1.29	0.34	51.88	11	16.82	22	2.09	12.40	100
4	0.04	964	16.06	235.00	3	0.12	0.90	1.99	0.49	54.20	16	23.26	31	3.06	13.15	100
4	0.03	1395	31.92	368.67	3	0.14	1.25	2.85	0.66	52.82	26	32.07	41	2.90	9.04	100
4	0.02	2382	74.45	606.67	3	0.30	2.17	4.51	1.10	50.58	44	54.46	66	3.93	7.21	100
4	0.01	4475	189.34	1136.00	3	1.10	5.07	10.46	2.43	47.96	99	118.57	138	7.23	6.10	100
4	0.005	7949	503.37	2073.33	3	2.97	11.06	19.49	4.85	43.84	223	248.91	270	9.51	3.82	100
4	0.001	27890	3793.76	7266.00	3	39.78	76.39	114.12	20.34	26.63	1089	1124.70	1165	17.37	1.54	100
4	0	658825	*	*	1	62011.15	62011.15	62011.15	n/a	n/a	59960	59960.00	59960	n/a	n/a	1
5	0.5	4	0.01	3.00	3	0.00	0.02	0.05	0.01	56.31	1	1.00	1	0.00	0.00	100
5	0.25	26	0.05	8.67	3	0.01	0.08	0.19	0.05	59.27	2	2.30	3	0.46	19.92	100
5	0.2	50	0.10	14.33	3	0.00	0.11	0.28	0.07	60.11	3	3.76	5	0.72	19.22	100
5	0.15	101	0.39	28.33	3	0.01	0.15	0.39	0.09	59.85	4	4.89	7	0.86	17.57	100
5	0.1	264	1.31	53.00	3	0.02	0.21	0.46	0.12	56.31	5	5.84	10	1.01	17.25	100
5	0.05	900	11.66	185.00	3	0.10	0.62	1.44	0.33	52.70	13	15.91	21	1.65	10.37	100
5	0.04	1204	18.24	236.33	3	0.14	0.85	1.76	0.45	52.63	16	21.74	28	2.70	12.42	100
5	0.03	1811	32.68	339.33	3	0.21	1.19	4.40	0.65	54.62	24	29.79	38	2.82	9.47	100
5	0.02	3256	68.51	598.67	3	0.48	2.07	4.04	0.94	45.63	40	48.82	57	3.92	8.02	100
5	0.01	6585	217.98	1188.67	3	1.20	4.64	7.92	1.95	41.96	89	106.41	126	7.00	6.58	100
5	0.005	12028	564.90	2256.67	3	4.33	11.50	19.43	4.18	36.39	206	228.94	244	8.89	3.88	100
5	0.001	45599	4400.78	8645.33	3	66.72	100.02	133.16	18.82	18.81	1056	1123.06	1168	20.17	1.80	100
5	0	1240182	*	*	1	171331.44	171331.44	171331.44	n/a	n/a	88314	88314.00	88314	n/a	n/a	1
6	0.5	4	0.01	2.33	3	0.00	0.03	0.05	0.01	56.15	1	1.00	1	0.00	0.00	100
6	0.25	26	0.03	5.00	3	0.01	0.07	0.18	0.04	56.12	2	2.22	3	0.41	18.66	100
6	0.2	50	0.14	14.33	3	0.00	0.11	0.64	0.08	75.56	3	3.73	5	0.72	19.28	100
6	0.15	101	0.36	27.00	3	0.01	0.15	0.41	0.09	57.95	4	4.73	7	0.86	18.15	100
6	0.1	269	1.18	43.33	3	0.02	0.20	0.59	0.12	57.36	5	5.84	10	0.97	16.55	100
6	0.05	971	10.15	155.67	3	0.07	0.57	1.27	0.30	52.93	13	15.45	20	1.66	10.77	100
6	0.04	1318	16.74	223.33	3	0.11	0.80	1.64	0.40	50.33	17	21.15	29	2.22	10.47	100
6	0.03	2001	25.10	280.67	3	0.24	1.14	2.17	0.53	46.50	24	28.33	35	2.45	8.63	100
6	0.02	3705	68.32	538.33	3	0.51	1.93	3.87	0.84	43.62	38	46.23	60	3.58	7.74	100
6	0.01	7825	196.90	1061.00	3	1.39	4.44	8.35	1.77	39.80	78	95.96	112	6.31	6.58	100
6	0.005	14634	493.02	2031.00	3	5.10	10.83	17.33	3.52	32.50	186	206.63	225	7.91	3.83	100
6	0.001	57741	3880.13	7842.67	3	74.39	105.56	139.61	16.58	15.71	963	1015.85	1065	20.88	2.06	100
6	0	1321685	*	*	1	179456.00	179456.00	179456.00	n/a	n/a	80350	80350.00	80350	n/a	n/a	1

Table 6.18 Statistics about the U-CIT objects created for the $K_{weighted}$ coverage criterion, where the columns, respectively, report the coverage strengths, the weight cutoff values, the numbers of testable entities to be covered, and the average construction times (in seconds) as well as the average sizes of the U-CIT objects computed by the generate-and-cover and cover-and-generate constructors together with the minimum, maximum, standard deviation, and coefficient of variation statistics for the results obtained from the latter constructor. Furthermore, the number of times the experiments were repeated are given in the column "repeat count."

			generate-and-cover constructor			cover-and-generate constructor									
						time					size				
t	cutoff	no of entities	avg. time	avg. size	repeat count	min.	avg.	max.	sd.	cv.	min.	avg.	max.	sd.	cv.	repeat count
2	0.70	332	7.16	167.67	3	0.07	0.76	1.73	0.48	62.75	27	31.74	37	1.97	6.22	100
2	0.75	426	10.23	209.67	3	0.09	0.99	2.16	0.63	63.43	36	41.10	47	2.20	5.36	100
2	0.80	566	17.90	279.33	3	0.13	1.38	3.03	0.88	63.99	49	55.73	61	2.26	4.05	100
2	0.85	793	34.33	400.33	3	0.24	2.11	4.64	1.30	61.60	77	85.40	94	3.00	3.51	100
2	0.90	1185	63.95	574.67	3	0.46	3.33	7.03	1.99	59.87	126	133.28	141	3.60	2.70	100
2	0.95	2073	162.53	990.33	3	1.10	6.42	14.11	3.73	58.04	235	245.66	256	4.46	1.82	100
3	0.70	2171	100.70	777.33	3	0.49	2.73	5.86	1.54	56.41	100	111.32	120	4.15	3.73	100
3	0.75	2951	181.72	1090.00	3	0.89	4.14	8.51	2.25	54.34	150	164.80	178	5.04	3.06	100
3	0.80	4117	303.40	1488.67	3	1.59	6.60	12.70	3.40	51.50	227	242.96	263	6.28	2.58	100
3	0.85	6051	582.78	2164.00	3	3.40	11.20	21.29	5.25	46.83	362	384.02	405	8.09	2.11	100
3	0.90	9675	1291.66	3447.33	3	9.26	22.47	38.32	8.79	39.12	639	666.56	696	10.16	1.52	100
3	0.95	17826	3671.09	6332.00	3	34.85	63.21	94.30	17.61	27.86	1320	1347.32	1387	14.03	1.04	100
4	0.70	8037	493.52	2095.67	3	3.07	7.74	13.70	3.07	39.64	218	251.25	275	8.98	3.57	100
4	0.75	11240	850.59	2905.67	3	6.23	13.25	25.50	4.72	35.59	357	382.40	414	10.73	2.80	100
4	0.80	16252	1608.39	4222.33	3	12.85	23.91	37.19	7.35	30.76	568	605.22	645	13.96	2.31	100
4	0.85	24798	3177.50	6454.67	3	31.82	50.10	71.79	11.96	23.88	947	984.22	1039	19.26	1.96	100
4	0.90	40330	6942.08	10454.67	3	90.62	127.55	200.33	23.77	18.64	1675	1734.26	1816	24.83	1.43	100
4	0.95	79330	21111.57	20702.00	3	403.89	459.80	500.68	19.50	4.24	3768	3868.14	3923	29.77	0.77	100
5	0.70	18518	1070.09	3516.67	3	10.01	16.76	24.58	4.39	26.17	362	386.81	413	11.02	2.85	100
5	0.75	26467	1896.90	5018.33	3	20.37	31.69	46.81	7.37	23.24	573	608.41	662	15.30	2.51	100
5	0.80	39309	3540.80	7484.33	3	47.71	66.26	109.47	12.47	18.83	911	956.32	1003	19.17	2.00	100
5	0.85	60222	6776.82	11397.67	3	115.87	132.83	148.14	7.46	5.61	1499	1555.03	1612	25.24	1.62	100
5	0.90	101015	15938.67	19338.33	3	362.41	411.02	439.87	14.26	3.47	2751	2849.14	2937	37.23	1.31	100
5	0.95	211007	50420.67	40552.50	2	1977.17	2081.92	2156.12	39.50	1.90	6738	6857.15	6974	48.97	0.71	100
6	0.70	29272	1380.18	4027.33	3	18.73	27.23	40.74	5.67	20.81	442	475.20	502	12.64	2.66	100
6	0.75	42694	2511.12	5857.00	3	40.92	54.86	90.64	9.36	17.06	708	743.64	789	16.65	2.24	100
6	0.80	63498	4650.73	8831.33	3	86.44	102.17	113.87	5.70	5.58	1065	1113.34	1156	20.65	1.85	100
6	0.85	98886	9152.57	13601.33	3	227.07	260.64	288.15	10.86	4.17	1765	1853.89	1936	32.88	1.77	100
6	0.90	169254	22546.59	23611.67	3	775.99	846.07	896.63	25.20	2.98	3359	3449.67	3540	37.66	1.09	100
6	0.95	368650	75936.20	52238.00	1	4482.59	4611.18	4785.40	61.42	1.33	8495	8627.77	8794	57.54	0.67	100

structors, especially the cover-and-generate constructor, were scaled to obtain full coverage under K_{seen} for various values of t up to and including 6, even when the frequency cutoff was 0. As a matter of fact, we chose to stop at the strength of 6, because, in the presence of 9 options, increasing the strength any further was quickly becoming exhaustive testing, which, in this context, is the same as testing all the distinct configurations seen in the field.

We then observed that the cover-and-generate constructor performed generally better than the generate-and-cover constructor in reducing both the covering array sizes and construction times. More specifically, the cover-and-generate constructor reduced the sizes by an average of 65.62%, 82.55%, 86.35%, 88.25%, and 88.16% while at the same time reducing the construction times by an average of 96.35%, 97.20%, 97.88%, 97.72%, and 97.32% when $t = 2$, 3, 4, 5, and 6, respectively. We, therefore, focus on the results obtained from the cover-and-generate constructor in the remainder of this section.

When $t \leq 3$ and cutoff=0, i.e., when all the t-tuples seen in the field are required to be covered, the sizes of the U-CIT objects generated by the cover-and-generate constructor, were smaller than the number of distinct configurations seen in the field, i.e., 37503. More specifically, the average sizes were 3625.24 and 24971.00 with the average construction times of 205.55 and 7216.62 seconds for $t = 2$ and 3, respectively (Table 6.17). When $t > 3$ and cutoff=0, however, the U-CIT objects had more than 37503 configurations, on average (Table 6.17).

In reality, when testing all the t-tuples seen in the field is still not practical due to the cost, the cutoff parameters of the usage-based coverage criteria can be utilized to select a weighted fraction of the tuples for testing. For example, when the frequency cutoff was set to 0.001 with K_{seen}, i.e., when the tuples that appeared in at least one thousandth of the configurations seen in the field were to be covered, the average sizes of the U-CIT objects became 327.31, 817.66, 1124.70, 1123.06, and 1015.85 when $t = 2$, 3, 4, 5, and 6, respectively.

All the results we obtained under different coverage strengths and cutoff values can be found in Table 6.17. For a fixed strength, as the cutoff increased, the number of testable entities as well as the size of the U-CIT objects tended to decrease. For example, when $t = 6$, the average sizes of the U-CIT objects were 1015.85, 206.63, 95.96, 46.23, and 5.84 for cutoff=0.001, 0.005, 0.01, 0.02, and 0.1, respectively. For a fixed cutoff, as the strength increased, on the other hand, although the number of testable entities to be covered increased, this did not necessarily cause an increase in the sizes of the U-CIT objects computed. For example, when cutoff=0.005, the average size of the U-CIT objects was 228.94 for $t = 5$, but 206.63 for $t = 6$. We

believe that this was because covering a frequently appearing t-tuple covers multiple frequently appearing t'-tuples, where $t' < t$. Thus, covering higher strength tuples may help reduce the number of test cases needed by covering more required tuples per test case. Regarding the construction times, except for the experimental setups, in which cutoff=0, all the constructions times were under 106 seconds, with a majority of them being under 12 seconds, on average.

Evaluating the $K_{weighted}$ coverage criterion. Table 6.18 summarizes the results we obtained from the experiments, in which we used the $K_{weighted}$ coverage criterion.

As was the case with K_{seen}, the cover-and-generate constructor, compared to the generate-and-cover constructor, computed smaller covering arrays at a fraction of the cost. More specifically, the cover-and-generate constructor reduced the sizes by an average of 77.39%, 80.93%, 83.29%, 83.09%, and 80.29% while at the same time reducing the construction times by an average of 94.94%, 98.20%, 98.00%, 95.88%, and 92.00%, when $t = 2$, 3, 4, 5, and 6, respectively. We, therefore, focus on the results obtained from the cover-and-generate constructor in the remainder of this section.

We observed that the sizes of all the U-CIT objects we computed for the study, were profoundly smaller than the number of distinct configurations observed in the field (Table 6.18). More specifically, the maximum average size was 8627.77, which occurred when $t = 6$ and the weighted cutoff was 0.95. That is, to cover 95% of the most frequently appearing t-tuples for all $1 \leq t \leq 6$ in a weighted manner as described in Definition 28, an U-CIT object of size 8627.77 was needed, on average.

For a fixed strength, as the cutoff decreased, the number of testable entities to be covered as well as the size of the U-CIT objects tended to decrease. For example, when $t = 6$, the sizes of the U-CIT objects for cutoff=0.95, 0.90, 0.85, 0.80, 0.75, and 0.70, were, respectively, 8627.77, 3449.67, 1853.89, 1113.34, 743.64, and 475.20 (Table 6.18). Similarly, for a fixed cutoff, as the coverage strength decreased, both the number of testable entities to be covered as well as the size of the U-CIT objects tended to decrease. For example, when cutoff=0.95, the average sizes were 8627.77, 6857.15, 3868.14, 1347.32, and 245.66 for $t = 6$, 5, 4, 3, and 2, respectively. Last but not least, the maximum average construction time in all the experiments was 4611.18 seconds, which happened when $t = 6$ and $cutoff = 0.95$. A majority of the construction times (79.4%) were, on the other hand, under 150 seconds (Table 6.18).

Evaluating sensitivity to the order of processing. Tables 6.17 and 6.18 report the minimum, maximum, standard deviation, and coefficient of variation (i.e., the ratio of the standard variation to the mean, in short CV) results obtained from the

cover-and-generate constructor by repeating the experiments 100 times (except for the experimental setups where the frequency cutoff was 0 and $t > 2$, which were repeated only once due to their costs). Clearly, the performance of the cover-and-generate constructor can be affected by the order, in which the testable entities are processed. Consequently, in the absence of any knowledge regarding a favorable order (or a partial order), a random order can be used by shuffling the entities to be covered before they are fed to the constructor. This process can further be repeated multiple times in an attempt to generate smaller CIT objects at the cost of increased construction times.

6.3.5 Discussion

Note that the maximum coverage strength that can be used with the $K_{weighted}$ coverage criterion is the number of configuration options that the system under test has. Therefore, $K_{weighted}$, in a sense, offers a solution to an important, but still an open question of how to determine the coverage strength in CIT, by automatically determining strength based on usage statistics. That is, the strength of a tuple to be covered by $K_{weighted}$, essentially depends on how frequently the tuple appears in the field. Consequently, the strength may vary across the test space. This is different than variable strength covering arrays (Cohen, Gibbons, Mugridge, Colbourn & Collofello, 2003), because in variable strength covering arrays, the strengths are determined a priori and they vary at the level of option combinations. In the $K_{weighted}$ coverage criterion, on the other hand, the strengths vary at the level of option setting combinations and they are determined based on usage statistics. Therefore, no strength needs to be determined beforehand.

Note that U-CIT does not aim to replace standard covering array constructors. We, indeed, don't see much value in using U-CIT to compute the same CIT objects that the existing CIT constructors compute, as the generalized U-CIT constructors may not be as efficient and as effective as their specialized counterparts. For example, when we used the cover-and-generate constructor to compute standard covering arrays for the configuration space models used in the experiments, the aforementioned U-CIT constructor generated a 2-way standard covering array of size 87586 in 32263.90 seconds (vs. a 2-way covering array of size 86241 generated in 100 seconds by ACTS) and failed to generate a 3-way standard covering array within a day, after which we stopped the constructor (vs. a 3-way covering array of size 13283730 generated in 32 minutes by ACTS).

The point we want to emphasize, however, is that even if U-CIT was able to reduce the sizes by half and did so in seconds, it would still not be feasible at all (for the consumer company, for which we carried out the study) to run all the test cases selected. Therefore, the coverage criteria needed to be changed. However, existing constructors, as they are, could not take advantage of these new criteria, which required fewer tuples to be covered.

Last but not least, it seems that for the usage-based CIT problem, it may actually be possible to modify an existing constructor. This, however, requires that the source code of the constructor is available, the code is reversed engineered, and a modification strategy is implemented, tested, and maintained. Note, however, that even if this was possible, these modifications would be of little help (or of no help at all) to compute the structure-based and order-based CIT objects we discussed in Chapter 6.1 and Chapter 6.2, respectively. Consequently, another set of modifications would be required to compute the structure-based CIT objects, such that the values of model parameters can be expressed as arbitrarily complex Boolean expressions. Similarly, different set of modifications would be required to compute the order-based CIT objects, such that the reachability restrictions imposed by a graph-based model can be expressed as constraints to cover various orders of nodes. As a matter of fact, we don't know how these modifications can be made without the solution quickly converging to U-CIT. This is exactly why U-CIT aims to eliminate the need of modifying existing constructors or developing specialized constructors, by generalizing the construction of CIT objects as much as possible.

7. HINTS

In this chapter, we present a new approach to improve the efficiency of the U-CIT constructors. A U-CIT constructor can be used as long as the entities to be covered are expressed as constraints and an appropriate solver is provided to determine if a given set of entities can be tested together in a single test case, i.e., if the respective constraints can be satisfied together. However, this may get quite expensive since finding an efficient solver for the given constraints may not be trivial in general. On the other hand, efficiency of U-CIT constructor highly depends on the number constraint problems solved by the constructor. Thus, we believe that the performance can be significantly increased by decreasing the number of attempts to solve constraint problems with constraint solvers.

To this end, we extend Unified Combinatorial Interaction Testing with a new mechanism, which we call "hints". The idea behind using hints stems from an observation of ours: Testable entities to be covered are typically composed of the same set of sub-entities, e.g., the same conjuncts appear in multiple testable entities. Therefore, in the processes of computing U-CIT objects, the same constraints are often solved multiple times. Consequently, capturing the relationships between these recurring constraints (i.e., sub-entities) in the form of hints can improve the efficiency of U-CIT constructors by reducing the number of times the solver is called and/or by calling the solver with simpler constraints.

As an example, consider a software system modelled as a directed acyclic graph and the requested coverage criterion to be achieved is to cover every possible 3-length node orderings. Moreover, assume that this graph has only two possible paths from the source node to sink node, meaning that if there is a solution to cover any possible node ordering, it can be only one of those paths. Then, the problem to be solved becomes that given a set of 3-length node orderings, can we find a path covering all those orderings from the source node to sink node? Solving these type of constraint problems using constraint solvers repeatedly may consume unnecessary resources when we consider how easy the problem is.

To overcome this issue, we define 2 type of hint sets for each entity, namely *contains* and *conflicts*. The contains set contains the sub-parts of entities in which satisfying all those sub-parts is enough to tell the respective entity is satisfiable with the constraints in the cluster. For instance, in order to show that the node ordering $[n_1, n_2, n_3]$ is satisfiable with a cluster, the sub-parts, e.g., $[n_1, n_2]$ and $[n_2, n_3]$, of the ordering needs to be satisfied by the cluster. Then, for this example, the contains set becomes: $\{[n_1, n_2], [n_2, n_3]\}$. On the other hand, the conflicts set contains the negated sub-parts of entities expressing that if at least one of those parts is satisfiable with the cluster, then, the respective entity is unsatisfiable. As an example, for the same node ordering $[n_1, n_2, n_3]$, the conflicts set becomes $\{[n_2, n_1], [n_3, n_2], [n_3, n_1]\}$. Note that, even one of the orderings from conflicts set becomes satisfiable with the cluster, the respective entity can not be added to the cluster.

The contains set, therefore, may decide whether an entity can be satisfiable with a set of constraints, i.e., can be added to a cluster, on the other hand, the conflicts set may decide whether the entity can not be satisfiable with the given set of constraints. However, it is not guaranteed that either contains or conflicts set can always decide the satisfiability or unsatisfiability of an entity with the cluster. In this situations, standard U-CIT procedure is followed by calling the "solver" function.

U-CIT constructor does not need to interpret the semantics of any constraints. Thus, instead of giving sub-parts of entities as sub-constraints, we assign them a symbolic parameters. As an example, the conflicts set $\{[n_2, n_1], [n_3, n_2], [n_3, n_1]\}$, can be represented with the parameters: $\{h_1, h_2, h_3\}$, i.e., for each sub-parts, a unique parameter h_i is defined.

To better explain how hints approach work and compare it with the cover-and-generate constructor (Algorithm 1) given in Chapter 5, we replicate the same algorithm in this chapter as Algorithm 4.

In the next sections, to illustrate how the hint approach works, we, first, express a standard t-way covering array (Chapter 2.1.1) as a U-CIT problem and compute the covering array using cover-and-generate constructor. Then, on the same example, we describe how the hints can be defined for the standard covering arrays and use the cover-and-generate constructor extended with hint mechanism (Algorithm 5) to compute the requested covering array. Finally, we apply the hint approach on two different domains to further both show the flexibility of the U-CIT and efficiency of hints approach.

Algorithm 4 The cover-and-generate constructor for computing U-CIT objects

Input: A test space model $M =< P, D, C >$
Input: A set of testable entities E to be covered
Output: An U-CIT object T

```
1: S ← {}
2: for each testable entity e ∈ E do
3:     accommodated ← false
4:     for each E' ∈ S do
5:         if satisfiable(e ∧ ⋀_{e'∈E'} e' ∧ C) then
6:             E' ← E' ∪ {e}
7:             accommodated ← true
8:             break
9:         end if
10:    end for
11:    if not accommodated then
12:        S ← S ∪ {{e}}
13:    end if
14: end for
15:
16: T ← {}
17: for each E' ∈ S do
18:    T ← T ∪ solve(C ∧ ⋀_{e'∈E'} e')
19: end for
20: return T
```

7.1 Expressing Standard Covering Arrays as U-CIT Problem

We express a standard 2-way covering array for a given configuration space model as a U-CIT object and then use the cover-and-generate constructor (Algorithm 4) to compute the requested object.

Suppose that the system under test has three configuration options $\{o_1, o_2, o_3\}$, each of which can take on either $true$ or $false$. Two system-wide constraints are defined on these options: $(o_2 = true) \implies (o_3 = true)$, i.e., if o_2 is $true$, then o_3 must be $true$, and $\neg(o_1 = true \wedge o_3 = false)$, i.e., the combination $(o_1 = true, o_3 = false)$ is invalid. Suppose further that all valid 2-tuples are required to be covered, i.e., essentially a standard 2-way covering array is needed.

Boolean logic can be used to define the U-CIT object to be computed in this scenario, where each configuration option is represented as a boolean variable and the system-wide constraints are represented as boolean constraints. Consequently, the U-CIT model can be defined as $M =< P, D, C >$, where $P = \{o_1, o_2, o_3\}$,

Table 7.1 U-CIT testable entities to be covered for a 2-way standard covering array.

(o_1, o_2)	(o_1, o_3)	(o_2, o_3)
$e_1 : \neg o_1 \wedge \neg o_2$	$e_5 : \neg o_1 \wedge \neg o_3$	$e_8 : \neg o_2 \wedge \neg o_3$
$e_2 : \neg o_1 \wedge o_2$	$e_6 : \neg o_1 \wedge o_3$	$e_9 : \neg o_2 \wedge o_3$
$e_3 : o_1 \wedge \neg o_2$		
$e_4 : o_1 \wedge o_2$	$e_7 : o_1 \wedge o_3$	$e_{10} : o_2 \wedge o_3$

$D = \{\{true, false\}, \{true, false\}, \{true, false\}\}$, and $C : (o_2 \implies o_3) \wedge \neg(o_1 \wedge \neg o_3)$.

Each U-CIT testable entity then naturally corresponds to a valid 2-tuple, which is expressed as a boolean constraint. Table 7.1 presents all the U-CIT testable entities $E = \{e_1, \ldots, e_{10}\}$ to be covered. For example, the testable entity $e_2 : (\neg o_1 \wedge o_2)$ represents the 2-tuple $(o_1 = false, o_2 = true)$. Note that two 2-tuples are excluded from Table 7.1 as they are invalid given the model constraint C. One of these tuples is $(o_2 = true, o_3 = false)$. It is an invalid tuple because $C \wedge (o_2 \wedge \neg o_3)$, i.e., $(o_2 \wedge \neg o_3) \wedge (o_2 \implies o_3) \wedge \neg(o_1 \wedge \neg o_3)$, is not satisfiable.

Assuming that the testable entities in Table 7.1 are processed in the order $e_1, \ldots, e_{10}$, our cover-and-generate constructor (Algorithm 4) proceeds as follows: First, $e_1 : (\neg o_1 \wedge \neg o_2)$ is processed. Since the pool S is initially empty (line 1), a new cluster $E_1 = \{e_1\}$ is created and S is populated with E_1, i.e., $S = \{E_1\}$ (line 12). Then, $e_2 : (\neg o_1 \wedge o_2)$ is processed. Since $e_1 \wedge e_2 \wedge C$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge o_2) \wedge (o_2 \implies o_3) \wedge \neg(o_1 \wedge \neg o_3)$, is not satisfiable (line 5), e_2 cannot be placed in E_1. So, a new cluster $E_2 = \{e_2\}$ is created and S is updated to $\{E_1, E_2\}$ (line 12). After processing the testable entities e_3 and e_4, S will have four clusters E_1, E_2, E_3, and E_4, having e_1, e_2, e_3, and e_4, respectively. Next, $e_5 : (\neg o_1 \wedge \neg o_3)$ is processed. As $e_1 \wedge e_5 \wedge C$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge \neg o_3) \wedge (o_2 \implies o_3) \wedge \neg(o_1 \wedge \neg o_3)$, is satisfiable (line 5), e_5 is included in E_1 (line 6). After processing all the remaining testable entities in Table 7.1, we will have the four clusters given in the first column of Table 7.2.

For each cluster in $S = \{E_1, E_2, E_3, E_4\}$, we then generate a U-CIT test case by satisfying the constraints included in the cluster together with the model constraint

Table 7.2 A U-CIT object (second column) created for the set of satisfiable clusters $S = \{E_1, E_2, E_3, E_4\}$ (first column) obtained for the testable entities in Table 7.1

	2-way covering array		
$S = \{E_1, E_2, E_3, E_4\}$	o_1	o_2	o_3
$E_1 = \{e_1, e_5, e_8\}$	false	false	false
$E_2 = \{e_2, e_6\}$	false	true	true
$E_3 = \{e_3, e_7, e_9\}$	true	false	true
$E_4 = \{e_4, e_{10}\}$	true	true	true

Algorithm 5 Extended cover-and-generate constructor with hints for computing U-CIT objects

Input: A test space model $M =< P, D, C >$
Input: A set of testable entities E to be covered
Output: An U-CIT object T

```
1:  S ← {}
2:  for each testable entity e ∈ E do
3:      S' ← {}
4:      accommodated ← false
5:      for each E' ∈ S do
6:          decision ← hint(E', e)
7:          if decision == contains then
8:              E' ← E' ∪ {e}
9:              accommodated ← true
10:             break
11:         else if decision ≠ conflicts then
12:             S' ← S' ∪ E'
13:         end if
14:     end for
15:
16:     if accommodated then
17:         continue
18:     end if
19:
20:     for each E' ∈ S' do
21:         if satisfiable(e ∧ ⋀_{e'∈E'} e' ∧ C) then
22:             E' ← E' ∪ {e}
23:             E'_contains ← E'_contains ∪ e_contains
24:             accommodated ← true
25:             break
26:         end if
27:     end for
28:     if not accommodated then
29:         E'' ← {e}
30:         E''_contains ← e_contains
31:         S ← S ∪ E''
32:     end if
33: end for
34:
```

C (lines 16-19). For example, for E_1, solving $e_1 \wedge e_5 \wedge e_8 \wedge C$ produces the test case $(o_1 = false, o_2 = false, o_3 = false)$. Processing all the clusters would then generate the U-CIT object given in the second column of Table 7.2 (line 20), which is indeed a standard 2-way covering array.

7.2 U-CIT with Hints

In this section, first, we discuss the motivation behind hint approach using the example given in the previous section, then, we explain how we embedded the proposed approach into U-CIT by applying the approach on the same example.

To explain our reasoning, lets count the number of solver calls (line 5 of Algorithm 4) made in the previous example. For e_1, line 5 is not executed, hence there is no solver call for the entity. For e_2, the algorithm attempts to accommodate the entity to E_1 cluster and then create a second cluster E_2. Thus, there is 1 solver call. In the end, 18 solver call needs to be done (excluding lines 16-20) to accommodate every entity to a cluster.

However, we believe that to compute a 2-way standard covering array having only 3 binary configuration options, there would not need to make 20 solver calls when we consider how easy the problem is. Note that the number of solver calls depends on the number of entities exponentially. That is, an increase in the number of entities may increase the number of solver calls dramatically. Thus, the number of solver calls may play an important role in the cost of covering array computation.

With help of hints (contains and conflicts sets), on the other hand, the tool could have easily decided the satisfiability or unsatisfiability of several entities to avoid making unnecessary solver calls.

To illustrate the usage of hints, we use the same example given in the previous section. The contains set of an entity is defined as a set of symbols representing each option-setting pair $< o_i, v_{ij} >$ of the respective entity. The symbols for each possible option-setting pair are given in Table 7.3 and the contains set of all entities are given in the second column of Table 7.4. For example, for the entity $e_3 : o_1 \wedge \neg o_2$, the contains set is formed with symbolic representations of $< o_1, true >$ and $< o_2, false >$, i.e., h_1 and h_5. On the other hand, the conflict set of an entity is constructed by enumerating all possible $< o_i, v_{ik} >$ pairs for each pair $< o_i, v_{ij} >$ from entity where $k \neq j$. That is, we enumerate all possible negotiations for each option-setting pairs to ensure that the entities which contain conflicting option-pairs, can not be con-

Table 7.3 Hint symbols for each option and setting pair.

options	true	false
o_1	h_1	h_4
o_2	h_2	h_5
o_3	h_3	h_6

Table 7.4 Contains and conflicts set of entities.

entity	contains	conflicts
$e_1 : \neg o_1 \wedge \neg o_2$	h_4, h_5	h_1, h_2
$e_2 : \neg o_1 \wedge o_2$	h_4, h_2	h_1, h_5
$e_3 : o_1 \wedge \neg o_2$	h_1, h_5	h_4, h_2
$e_4 : o_1 \wedge o_2$	h_1, h_2	h_4, h_5
$e_5 : \neg o_1 \wedge \neg o_3$	h_4, h_6	h_1, h_3
$e_6 : \neg o_1 \wedge o_3$	h_4, h_3	h_1, h_6
$e_7 : o_1 \wedge o_3$	h_1, h_3	h_4, h_6
$e_8 : \neg o_2 \wedge \neg o_3$	h_5, h_6	h_2, h_3
$e_9 : \neg o_2 \wedge o_3$	h_5, h_3	h_2, h_6
$e_{10} : o_2 \wedge o_3$	h_2, h_3	h_5, h_6

tained by the cluster. For the same entity $e_3 : o_1 \wedge \neg o_2$, as an example, the conflict set is formed with symbolic representations of $< o_1, false >$ and $< o_2, true >$ from Table 7.3, i.e., h_4 and h_2. The conflicts sets for all the entities are given in the last column of Table 7.4.

We present our proposed approach U-CIT with hints in Algorithm 5. Compared to Algorithm 4, Algorithm 5, before making a solver call (line 21 of Algorithm 5), checks to see whether the entity can be accommodated to any cluster using hints approach (lines 5-14 of Algorithm 5). If the hint approach can not tell anything about the entity, then, we use the same cover-and generate constructor.

In Algorithm 5, the clusters (e.g., E_i) needs to hold the hint information, more specifically contains set. Each time an entity is accommodated to a cluster (lines 21-26 and 28-32 of Algorithm 5), the symbols from contains set of entity are added to contains set of cluster. Then, to decide that an entity can be accommodated to a cluster, all symbols within entity contains set must be present in the contain set of the cluster. On the other hand, to decide that an entity cannot be accommodated to a cluster, at least one of the symbols from conflict set of the entity must be present in the contains set of cluster.

Revisiting our running example given in Section 7.2, assuming that the testable entities in Table 7.4 are processed in the order $e_1, \ldots, e_{10}$, the new constructor (Algorithm 5) proceeds as follows: First, $e_1 : (\neg o_1 \wedge \neg o_2)$ is processed. Since the pool S is initially empty, the constructor skips the hint part (lines 4-18) and creates a new cluster with e_1, $E_1 = \{e_1\}$. All symbols from contains set of e_1 are added to contains set of cluster $E_{1contains}$=$\{h_4, h_5\}$ (line 30). Then, $e_2 : (\neg o_1 \wedge o_2)$ is processed. The algorithm checks whether all symbols within the contains set of entity $\{h_4, h_2\}$ are present in the contains set of cluster: $\{h_4, h_5\}$. Since not all of them are present, the contains set did not help in this case. Then, we check whether any symbol within the conflict set of entity $\{h_1, h_5\}$ is present in the contains set of cluster. Since h_5 is present, we can directly say that e_2 can not be placed in cluster E_1 without

Table 7.5 Clusters covering each entity given in Table 7.4 and their contains sets.

$S = \{E_1, E_2, E_3, E_4\}$	contains sets
$E_1 = \{e_1, e_5, e_8\}$	h_4, h_5, h_6
$E_2 = \{e_2, e_6, e_{10}\}$	h_2, h_3, h_4
$E_3 = \{e_3, e_7, e_9\}$	h_1, h_3, h_5
$E_4 = \{e_4\}$	h_1, h_2

making a solver call. Note that even though contains set can not tell anything for this entity, with the help of conflict set we can skip the solver call part (lines 20-27). Therefore, a new cluster $E_2 = \{e_2\}$ is created with $E_{2contains}$=$\{h_4, h_5\}$ and S is updated to $\{E_1, E_2\}$ (lines 28-32). After processing the testable entities in the order e_3, e_4, e_5, e_6, and e_7 with the help of conflict sets, S will have four clusters: E_1=$\{e_1, e_5\}$, E_2=$\{e_2, e_6\}$, E_3=$\{e_3, e_7\}$, and E_4=$\{e_4\}$, and contains set of clusters becomes $E_{1contains}$=$\{h_4, h_5, h_6\}$, $E_{2contains}$=$\{h_2, h_3, h_4\}$, $E_{3contains}$=$\{h_1, h_3, h_5\}$, and $E_{4contains}$=$\{h_1, h_2\}$.

Next, $e_8 : \neg o_2 \wedge \neg o_3$ is processed. There are 2 symbols in the contains set of e_8: h_5 and h_6, which are both contained by the contains set of E_1. Thus, the entity can be accommodated to cluster E_1 (lines 7-10). Next, $e_9 : \neg o_2 \wedge o_3$ is processed. In the same manner, both symbols from contains set of e_9 is contained by the E_2. Thus, e_9 is accommodated to E_2. After all entities are processed, the clusters and their contains sets become as in Table 7.5.

In total, Algorithm 4 attempts to make 18 solver calls, whereas the Algorithm 5 using hint approach makes only 3 solver calls, proving the efficiency of the approach even on an easy covering array computation problem.

In the following sections, to evaluate hint approach, we carry out two case studies. In the first study (Chapter 7.3), we compute structure-based CIT objects similar to Chapter 6.1, however, in this work, we compute condition coverage-adequate objects (Chapter 2.2.2) instead of decision coverage-adequate objects to further show the flexibility of the approach. In the second study (Chapter 7.4), we compute sequence covering arrays 2.1.2, which are well-known combinatorial objects in CIT for mostly testing event-driven based systems.

All the experiments conducted in this chapter, unless otherwise stated, were repeated 3 times and carried out on Google Cloud using Intel Xeon CPU 2.30GHz machine with 4 GB of RAM, running 64-bit Ubuntu 18.04 as the operating system. Furthermore, we used OR-Tools (Perron & Furnon, 2019) as the constraint solver in the U-CIT constructor. In all the experiments we put a 5 hours (18000 secs.) time threshold.

Table 7.6 Information about the subject applications.

sut	version	description	actual options	virtual options	valid 1-combs	valid 2-combs	valid 3-combs
mpsolve	2.2	Mathematical solver	14	4	30	296	1104
dia	0.96.1	Diagramming application	15	11	40	694	6942
irissi	0.8.13	IRC client	30	11	72	2230	39492
xterm	2.4.3	Terminal emulator	38	31	84	4088	137844
parrot	0.9.1	Virtual machine	51	29	149	10264	424810
gimp	3.2.5	Vector graphics editor	79	28	231	26674	2011858
pidgin	2.4.0	IM	53	43	167	13919	729783
python	2.6.4	Programming language	68	49	178	16322	981792
xfig	2.6.8	Graphics manipulator	79	48	212	17798	874930
vim	7.3	Text editor	79	49	231	26647	2002485
sylpheed	2.6.0	E-mail client	84	48	300	43706	4057960
cherokee	1.0.2	Web server	97	28	238	28024	2075410

```
1  #ifdef (o1 && o2)
2    ...
3    #ifdef (o3 || o4)
4      ...
5    #endif
6  #endif
7
8  #ifdef (o5)
9    ...
10   #ifdef (o6)
11     ...
12   #endif
13 #endif
```

(a)

virtual options	conds.	virtual settings	hint symbols
vo_1	o_1	$e_1 : o_1$	h_1
	$\neg o_1$	$e_2 : \neg o_1$	h_2
	o_2	$e_3 : o_1 \wedge o_2$	h_3
	$\neg o_2$	$e_4 : o_1 \wedge \neg o_2$	h_4
	o_3	$e_5 : (o_1 \wedge o_2) \wedge o_3$	h_5
	$\neg o_3$	$e_6 : (o_1 \wedge o_2) \wedge \neg o_3$	h_6
	o_4	$e_7 : (o_1 \wedge o_2) \wedge \neg o_3 \wedge o_4$	h_7
	$\neg o_4$	$e_8 : (o_1 \wedge o_2) \wedge \neg o_3 \wedge \neg o_4$	h_8
vo_2	o_5	$e_9 : o_5$	h_9
	$\neg o_5$	$e_{10} : \neg o_5$	h_{10}
	o_6	$e_{11} : o_5 \wedge o_6$	h_{11}
	$\neg o_6$	$e_{12} : o_5 \wedge \neg o_6$	h_{12}

(b)

Figure 7.1 (a) An example set of preprocessor directives for a system with 6 compile-time configuration options and (b) set of entities for each possible condition for the given system.

7.3 Study 1: Structural Coverage

In Chapter 6.1, we computed structural U-CIT objects for the decision coverage, however, in this work, we not only compute U-CIT objects to get full coverage under condition coverage, also apply hints approach on the same subject applications to demonstrate the efficiency and effectiveness of hints approach.

In this section, as an example, we use the same set of preprocessor directives (Figure 4.1a) for a system with 6 compile-time configuration options for the condition coverage as given in Figure 7.1a. There are two virtual configuration options for this system: vo_1 representing the outer-most if-then-else directive between lines 1 and 6 and vo_2 representing the outer-most if-then-else directive between lines 8 and

13. Then, given a virtual configuration option, we define *virtual setting* as the each feasible outcome of every condition for the configuration options in the respective if-then-else directive. Virtual settings are expressed as a constraint, such that covering all of these virtual settings obtains a full coverage under CC. All of the virtual settings for both vo_1 and vo_2 are given in Figure 7.1b. For instance, the virtual option vo_1 in the example has 8 virtual settings: o_1, $\neg o_1$, $o_1 \wedge o_2$, $o_1 \wedge \neg o_2$, $(o_1 \wedge o_2) \wedge o_3$, $(o_1 \wedge o_2) \wedge \neg o_3$, $(o_1 \wedge o_2) \wedge \neg o_3 \wedge o_4$, and $(o_1 \wedge o_2) \wedge \neg o_3 \wedge \neg o_4$. The first two settings, as an example, are respectively for covering the $true$ and $false$ condition outcome of the o_1. In the same perspective, the third and fourth settings are respectively for covering the $true$ and $false$ condition outcome of the o_2. Note that in order to evaluate o_2, o_1 must be evaluated as $true$ due to short-circuit decision evaluation.

We, then, define an entity as a *t-combination*. That is, a combination of virtual settings for a combination of t distinct virtual options, which is expressed by joining the respective constraints with the AND logical operator. A t-combination is invalid, if the respective constraint is not satisfiable. An example of 2-combination for the virtual options vo_1 and vo_2 can be $(o_1 \wedge \neg o_2) \wedge (o_5 \wedge o_6)$ testing the interaction between the $false$ evaluation of o_2 and $true$ evaluation of o_6.

Finally, a number of configurations are selected to cover all valid t-way combinations of virtual option settings. The smaller the number of configurations selected, the better the approach is.

7.3.1 Expressing Problem as U-CIT

We have defined the U-CIT model as $M =< P, D, C >$, where P is the set of variables representing the actual configuration options; D is their respective domains, i.e., the settings that the actual configuration options can take on; and C is the model constraint (if any) invalidating certain combinations of option settings. Each U-CIT testable entity then naturally corresponded to a valid t-combination to be covered and each U-CIT test case naturally corresponded to a configuration, in which every actual configuration option has a valid setting.

7.3.2 Coverage Hints

For each possible outcome of conditions, we define a hint symbol as given in the fourth column of Figure 7.1b. Then, the contains set of an entity becomes the set of symbols which represents the conditions (the third column of Figure 7.1b) from t-way combination. On the contrary, the conflict set of an entity becomes the set of symbols which represents the negotiations of conditions from t-way combination. For example, consider the 2-way combination $< vo_1 = (o_1 \wedge o_2), vo_2 = (o_5 \wedge \neg o_6) >$ to cover $o_2 = true$ and $o_6 = false$ conditions. The contains and conflicts set for this 2-way combination becomes $\{h_3, h_{12}\}$ and $\{h_4, h_{11}\}$, respectively.

Note that, if both of the virtual settings $o_1 \wedge o_2$ and $o_5 \wedge \neg o_6$ are already satisfied by a cluster, then one can conclude that the given 2-way combination is already covered without further calling a constraint solver. In the same perspective, if any of the negotiation of virtual setting is already covered by the cluster, then, there is no way the respective 2-way combination can be covered.

7.3.3 Experiments

We used the same 12 subject applications studied in Chapter 6.1.2. Each application had a number of binary compile-time configuration options implemented by using preprocessor directives. Table 7.6 provides information about these subject applications. The columns of this table respectively present the subject applications, their versions and descriptions, the numbers of actual compile time options they have, the numbers of virtual options extracted, and the numbers of 1-, 2- and 3- combinations selected by our structure-based coverage criterion. Note that since we were not aware of any inter-option constraints for these subject applications, all possible combinations of option settings were considered to be valid.

To evaluate the proposed approach, we both compared the size of computed U-CIT objects and time to compute the U-CIT objects. In the overall evaluation, we only compared the experimental setups in which both methods have results, i.e., was not terminated due to the time threshold.

Table 7.7 presents all results. First column represents the subject applications used in the study, second and third columns compares the time to compute U-CIT objects in seconds, and 4th and 5th columns compares the sizes of computed U-CIT objects with and without using hints when t=1. The other columns compares the time and size results when t=2 and 3.

We observed that in 33% of the experimental setup (17% and 83% of them when

Table 7.7 Comparing structural U-CIT objects computed by using standard U-CIT constructor and U-CIT constructor with hint approach.

	t=1				t=2				t=3			
	time (secs.)		size		time (secs.)		size		time (secs.)		size	
SUT	with hints	wout hints	with hints	wout hints	with hints	wout hints	with hints	wout hints	with hints	wout hints	with hints	wout hints
cherokee	97.4	2.8	5.0	6.3	124.8	-	43.7	-	424.4	-	243	-
dia	2.6	0.3	4.3	4.0	4.0	17.4	19.0	21.3	6.8	859.1	67.7	72.7
gimp	85.7	2.7	5.0	5.7	115.7	16520.4	42.0	43.3	405.3	-	237.3	-
irissi	7.8	0.6	4.0	4.0	11.9	93.1	23.7	24.7	23.3	-	104.3	-
mpsolve	1.5	0.3	3.3	3.0	2.2	6.5	15.0	14.3	3.1	58.2	39.3	40.0
parrot	38.4	2.0	10.3	10.7	56.4	1589.6	59.0	58.3	158	-	334.7	-
pidgin	44.4	1.8	4.7	4.0	57.6	3281	34.0	34.0	162.7	-	168.7	-
python	52.7	2.0	4.7	4.0	70.5	3756.3	36.3	36.0	196.6	-	191.7	-
sylpheed	141.2	3.8	5.7	5.7	204.2	-	46.7	-	797	-	272.3	-
vim	85.8	2.6	6.0	6.0	112.7	13586.9	45.3	42.7	464.8	-	258.0	-
xfig	79.3	2.7	8.0	8.3	106.8	5552.2	53.3	53.7	257.6	-	300.3	-
xterm	10.1	0.8	4.0	4.0	13.8	183.2	20.0	20.0	27.8	-	74.0	-

t=2 and t=3, respectively), the traditional U-CIT constructor (without hint) failed to generate U-CIT objects in the given timeout.

In the comparable results, where both approaches have results, the average time to compute an U-CIT object was 50.4 and 1897.0 secs with hints and without hints, respectively. More specifically, when t=1, 2 and 3, for the former approach the results were 646.9, 551.6 and 9.9 seconds whereas for the latter approach the results were 22.4, 44586.6 and 917.3 secs, respectively. Note that, in all the results, except when t=1, timings to compute U-CIT objects much smaller for the approach using hints. For t=1, on the other hand, since entities are not formed as the conjunction of smaller parts, i.e., they are already a single component, hints did not help at all, as expected.

For the comparable size results, when t=1 and t=2, there were no significant difference. More specifically, when t=1, the average size results were 5.4 and 5.5, for the approaches with hints and and without hints, respectively. Moreover, when t=2, the results were both 34.8. However, we observed that when t=3, our proposed approach decreased the size of the U-CIT objects by 5.1% on the average. We believe the reason for this results is that when the entities can be accommodated with the help of contains set, we do not alter anything in the cluster. In other words, we mark that the entity is already covered with the current state of the cluster and the entity constraints are not added to the cluster as it is assumed to be already covered. However, when hints are not used, the entity may be accommodated to a different cluster and the complexity of constraint problem for that respective cluster would be increased even though it is already covered by any other cluster. Hence, the chance for that cluster to cover next entities might decrease.

7.4 Study 2: Computing Sequence Covering Arrays

In our earlier work (Mercan & Yilmaz, 2021), we described how to express a sequence covering array (Chapter 2.1.2) as a U-CIT problem. In this thesis, we use the same encoding both to express the the efficiency of hints approach and to compute higher strength sequence covering arrays.

7.4.1 Expressing Sequence Covering Arrays as U-CIT Objects

Sequence covering arrays can be expressed as a U-CIT problem as follows: Given a set of n events and a value of k, each event e_i $(1 \leq i \leq n)$ is represented by a parameter $e_i \in [1,n]$, the value of which determines the order of the respective event in a test case.

The U-CIT model is then specified as $M = < P, D, C >$, where $P = \{e_1, e_2, \ldots, e_n\}$ and $D = \{[1,n], \ldots, [1,n]\}$. Since, in a permutation of events, each event shall have a distinct order index, the model constraint C is defined as:

$$\forall e_i, e_j \in P, i \neq j \implies e_i \neq e_j. \tag{7.1}$$

For example, when $n = 4$, four parameters (e_1, e_2, e_3, e_4), each of which takes it value from the range of $[1,4]$, are defined. And, the assignments $e_1 = 4$, $e_2 = 3$, $e_3 = 1$, $e_4 = 2$, for instance, represents the test case $[e_3, e_4, e_2, e_1]$.

Given such a U-CIT model, a permutation of events $(e_{i_1}, e_{i_2}, \ldots, e_{i_k})$ to be tested corresponds to an entity to be covered, which is represented by the constraint:

$$e_{i_1} < e_{i_2} < \cdots < e_{i_k} \tag{7.2}$$

Table 7.8 represents a formulation for an (4, 3) sequence covering array. System model implicitly defines a valid space for test cases and each permutation of k distinct event is represented as a constraint.

Table 7.8 An example U-CIT formulation for (4, 3) sequence covering arrays

U-CIT Model
$e_1, e_2, e_3, e_4 \in [1, 4]$
$e_1 \neq e_2 \neq e_3 \neq e_4$

Entities
$e_1 < e_2 < e_3$
$e_1 < e_3 < e_2$
$e_2 < e_1 < e_3$
$e_2 < e_3 < e_1$
$e_3 < e_1 < e_2$
$e_3 < e_2 < e_1$
$e_1 < e_2 < e_4$
$e_1 < e_4 < e_2$
...
$e_4 < e_3 < e_2$

7.4.2 Coverage Hints

An important observation for a pair of events e_i and e_j in a test case is that there can be only 2 possible orderings. That is, either first e_i is executed then e_j, or vice versa. Thus, in general, every pair orderings will be present in a sequence covering array as the half of the number of test cases. In other words, the event pair orderings $[e_i, e_j]$ and $[e_j, e_i]$ will be covered almost equal number of times. We use this observation to define hint symbols for pair of event orderings since the more the entity components repeatedly occur in other entities, the better the hint approach works.

Then, for each event pair ordering e_i, e_j, a symbol h_i^j is defined as below:

$$h_i^j : e_i < e_j, \forall e_i, e_j \in P, i \neq j \tag{7.3}$$

h_i^j represents that e_i event is executed before e_j. Then, coverage hint symbols are all appended to each event ordering $[e_1, e_2, \ldots e_t]$ as below:

$$\begin{aligned} contains &: \{h_1^2, h_1^3, \ldots, h_{t-1}^t\} \\ conflicts &: \{h_2^1, h_3^1, \ldots, h_t^{t-1}\} \end{aligned} \tag{7.4}$$

Table 7.9 Comparing sequence covering arrays computed by using standard U-CIT constructor and U-CIT constructor with hint approach.

			construction time (secs.)		size	
events	k	entities	with hints	without hints	with hints	without hints
10	3	720	0.9	13.9	13.7	14.0
15	3	2730	5.0	171.4	15.7	16.3
20	3	6840	17.1	1046.0	18.3	18.3
25	3	13800	44.5	3919.9	19.7	20.3
30	3	24360	105.3	12462.9	21.0	21.0
40	3	59280	382.0	-	23.3	-
50	3	117600	1131.3	-	25.3	-
60	3	205320	2634.2	-	26.7	-
10	4	5040	3.0	845.1	72.0	73.3
15	4	32760	23.0	31154.2	96.7	98.0
20	4	116280	92.7	-	114.0	-
25	4	303600	267.0	-	127.3	-
30	4	657720	652.3	-	140.0	-
40	4	2193360	2512.6	-	158.0	-
50	4	5527200	7126.3	-	172.0	-
60	4	11703240	15720.7	-	183.7	-

7.4.3 Experiments

To evaluate the proposed approach, we have conducted a set of experiments. In the experiments, we have used the cover-and-generate constructor (Algorithm 4) to compute (n, k) sequence covering arrays for $n \in \{10, 15, 20, 25, 30, 40, 50, 60\}$ and $k \in \{3, 4\}$. Note that since (n, 2) sequence covering arrays can trivially be computed by randomly generating a permutation of the events and then adding the reverse of this permutation as the second test case, we did not experiment with $k = 2$.

Table 7.9 presents the results we obtained. The columns from left to right in the table indicate that the number of events used (i.e., n), the length of a sub-sequences to be covered, the average time (in seconds) it took to compute the respective sequence covering array, and the size of the arrays both for with hint and without hint. The symbol "-" marks experimental setups for which the respective constructor failed to compute sequence covering array within given time threshold.

In 56.3% (9 out of 16) of the experiment setups, standard U-CIT (without hints) failed to compute sequence covering arrays within given time threshold. For all the setups in which both approaches successfully computed U-CIT objects, on average, U-CIT with hints was 63, and 818 times faster than standard U-CIT for k 3 and 4, respectively. Moreover, U-CIT with hints managed to decrease the sizes by 0.3, and 1.3 configurations for k is 3 and 4, on average, respectively.

7.5 Discussion

In this chapter, we empirically demonstrated that the performance (i.e., the construction times) of U-CIT constructors can be significantly improved by using "hints". The idea behind using hints stems from a simple observation of ours: Testable entities to be covered are typically composed of the same set of sub-entities, e.g., the same conjuncts appear in multiple testable entities. Therefore, in the processes of computing U-CIT objects, the same constraints are often solved multiple times. Consequently, capturing the relationships between these recurring constraints (i.e., sub-entities) in the form of hints can improve the efficiency of U-CIT constructors by reducing the number of times the solver is called and/or by calling the solver with simpler constraints.

8. USER STUDIES

To further evaluate the proposed approach, we have also carried out user studies.

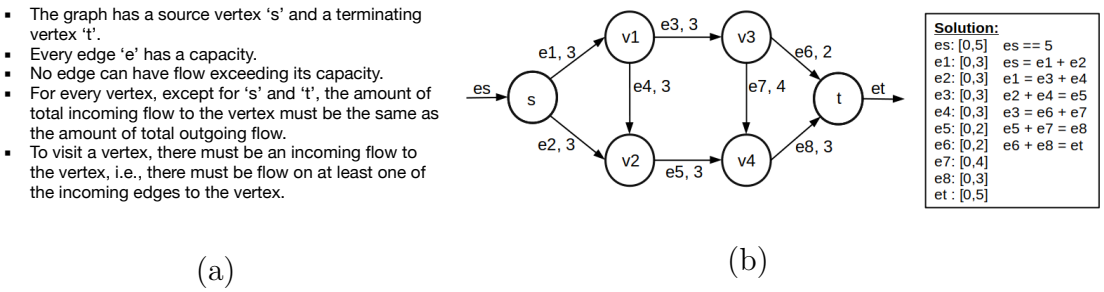
8.1 Study Setup

We asked the Junior, Senior, and graduate-level computer science students studying at Sabanci University whether they would take part in the study. A total of 13 graduate-level and 7 undergraduate-level students agreed to participate on a voluntary basis. Table 8.1 summarizes the demographic information about the participants. Note that students at Sabanci University study standard combinatorial interaction approaches at different levels and/or for different purposes in the Software Engineering (undergraduate level), Software Verification and Validation (undergraduate/graduate level), and Automated Debugging (graduate level) courses, which explains the participants knowledgable of CIT in Table 8.1.

The participants were first given a 1-hour lecture. In this lecture, after a brief introduction of how the study would be carried out, the basic concepts in constraint solving, such as Boolean logic and satisfiability, were discussed. Then, U-CIT was introduced. To this end, the definitions and the algorithms given in Chapter 5 were studied. Finally, a short tutorial on the U-CIT tool, which we had developed for

Table 8.1 Demographic information about the participants.

	undergraduate				graduate			
no of participants	7				13			
knowledgeable of CIT	**yes**		**no**		**yes**		**no**	
	1		6		10		3	
experience in programming (in years)	$\leq$ **2**	**3**	**4**	$\geq$ **5**	$\leq$ **2**	**3**	**4**	$\geq$ **5**
	2	3	2	0	0	2	5	6
experience in software testing (in years)	$\leq$ **2**	**3**	**4**	$\geq$ **5**	$\leq$ **2**	**3**	**4**	$\geq$ **5**
	7	0	0	0	11	1	0	1



(a) (b)

Figure 8.1 Explanations used for the second problem in the user study: (a) the description of the network flow problem and (b) an example network flow with incoming flow as 5 (i.e., $es = 5$) and its solution, where each edge has a label in the form of ex, c, indicating that c (except es and et) is the capacity of the edge ex.

Table 8.2 Problems used in the user studies.

	Problem 1	**Problem 2**	**Problem 3**
description	The same problem in Chapter 6.1 with 6 Boolean configuration options: $p_1, \ldots, p_6$.	The same problem in Chapter 6.2 with the graph given in Figure 8.2.	The same problem in Chapter 6.3 with 5 parameters: $p_1, p_2 : [1,3]$, $p_3 : [1,4]$, and $p_4, p_5 : [1,5]$.
entities	p_3 $p_1 \wedge p_2$ $\neg p_6 \wedge p4$ $p_2 \wedge \neg p_3 \wedge \neg p4$ $\neg(p_5 \vee \neg p_6)$	– t-orders – $[a_1, a_5]$ $[a_1, a_3, a_4]$ $[a_2, a_4, a_6]$ – consecutive-t-orders – $[a_4, a_6]$ $[a_1, a_3, a_4]$ $[a_3, a_4, a_6]$ – non-consecutive-t-orders – $[a_1, a_2]$ $[a_1, a_3, a_4]$ $[a_4, a_6]$	$(p_5 = 4)$ $(p_1 = 1, p_2 = 3)$ $(p_2 = 1, p_5 = 2)$ $(p_3 = 3, p_4 = 3, p_5 = 2)$ $(p_1 = 2, p_2 = 2, p_3 = 2, p_4 = 5, p_5 = 2)$

the study was given (see below for more information about this tool).

The participants, after taking the lecture, took part in the study at their spare times. To gain better insight, each participant carried out the study with one of the authors playing the role of an observer, sitting by the participant and taking notes. The participants were asked to think out loud as much as possible. When it was not clear for the observer what the participant was doing, the observer prompted the participant with questions, such as what do you want to do now? Is the output what you were expecting? What do you think what went wrong? etc.

Each participant was given with the same three problems. These problems were, indeed, the smaller instances of the very same problems we studied in Chapter 6.1, Chapter 6.2, and Chapter 6.3, respectively. For each problem, the participants were first asked to develop an U-CIT model $M =< P, D, C >$, then to express a number of U-CIT entities as constraints using M, and finally to generate an U-CIT object

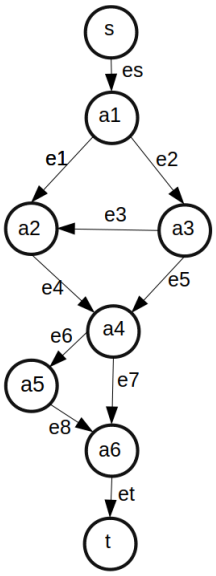


Figure 8.2 The graph-based model used in the user studies.

(by using the U-CIT constructor provided) to cover all of the given entities. The problems as well as the entities used in the study were given in Table 8.2.

Note that U-CIT is not a methodology for choosing the entities to be tested. It rather takes as input a set of entities to be tested. Therefore, the participants in the study were given with a set of entities to cover. For each problem, the entities were presented starting from the easier ones progressing to the more challenging ones. Furthermore, the number of entities was kept small not to tire the participants.

We designed the studies such that if a participant working on a problem got stuck after the first 10 minutes, the observer would remind the participant of the basic concepts that 1) the U-CIT model $M =< P, D, C >$ should define a set of parameters P and their domains D; 2) the model constraint C is a constraint that should be satisfied by all of the U-CIT test cases generated; and 3) the entities should be expressed as constraints over P.

Furthermore, for the second problem (Table 8.2), if the participant got stuck after the first 15 minutes, he/she was provided with a description of the network flow problem Bazaraa, Jarvis & Sherali (2011) given in Figure 8.1a. If the participant got stuck again 15 minutes after reading the description, a solution for the example flow problem given in Figure 8.1b was presented to the participant. Note that both the description in Figure 8.1a and the example in Figure 8.1b are general enough that they can be found in any textbook on the subject Bazaraa et al. (2011). Given these artifacts, the participants were still required to figure out how to express reachability in a graph as a flow problem and how to express different types of order-based entities (Chapter 6.1) as flow constraints.

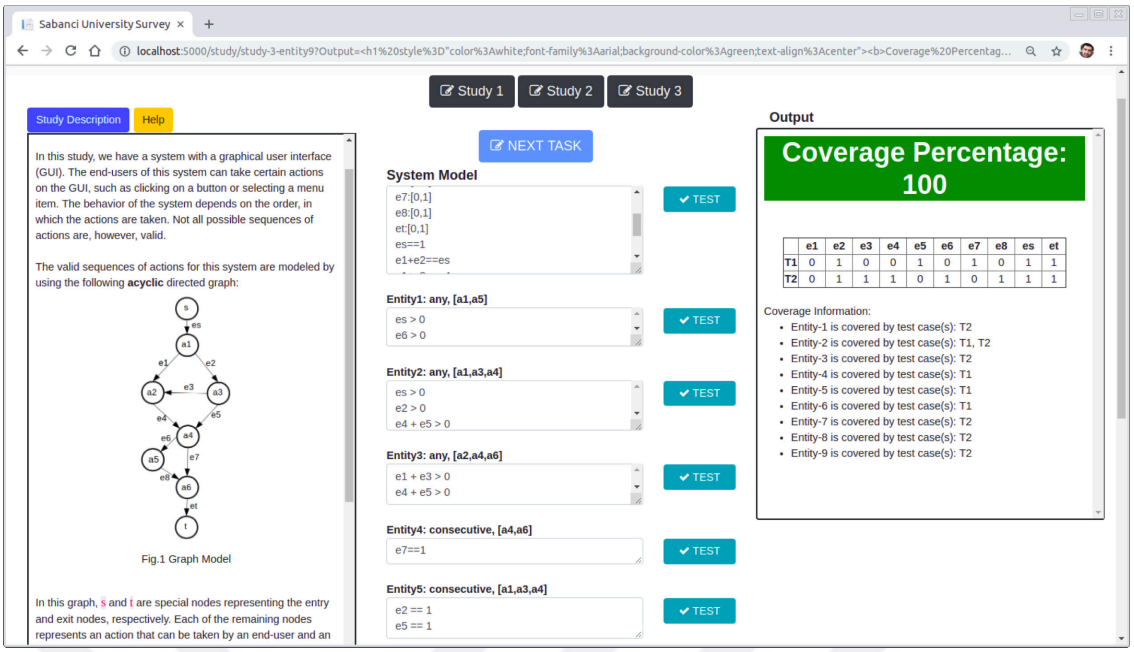


Figure 8.3 A screen dump of the tool we have developed for the user studies.

We did this because solving the aforementioned problem requires specific knowledge of network flow problems and not all participants might have had the right background. Therefore, by providing a general description of the network flow problems together with an example, we aim to answer the following question: Had the participants had a basic background information on network flow problems, could they have leveraged it in U-CIT to obtain full coverage under various order-based coverage criteria?

Last but not least, we have developed an U-CIT tool for the practitioners to use in the study. Figure 8.3 presents a screen dump taken from this tool. At a very high level, the tool has three frames. A description frame on the left, presenting the problem to be solved. An U-CIT frame in the middle where the participant expresses a solution to the given problem in U-CIT. An output frame on the right, which (among other things, see below for more information) presents the results obtained from the cover-and-generate constructor (Chapter 5.1) for the U-CIT formulation given in the middle frame.

The middle frame had a multi-line text field (model field) to express the U-CIT model and a multi-line text field (entity field) for expressing each entity to be covered. Each field had a “Test” button associated with it. When the Test button of the model field was clicked, the constraints entered for this field were fed to a constraint solver and the result was displayed in the output frame, allowing the participant to check whether the U-CIT model is capable of generating valid test cases. When the Test button of an entity field was clicked, on the other hand, the

Table 8.3 The exit survey used in the user studies. All the questions, except for the last two, were Likert scale questions. Questions 1-2 and 6-8 had the following answer options: *1 - strongly disagree*, *2 - disagree*, *3 - neutral*, *4 - agree*, and *5 - strongly agree*. And, questions 3-5 had the following answer options: *1 - very difficult*, *2 - difficult*, *3 - normal*, *4 - easy*, and *5 - very easy*. The last three questions (5-7) were open-ended questions.

no	question
Q1	I understand the following concepts:
a	constraints
b	satisfiability
c	unsatisfiability
Q2	I understand the following concepts:
a	U-CIT models
b	U-CIT entities
c	U-CIT test cases
d	U-CIT objects
Q3	For problem 1 – Difficulty of encoding:
a	U-CIT model
b	U-CIT entities
Q4	For problem 2 – Difficulty of encoding:
a	U-CIT model
b	U-CIT entities
Q5	For problem 3 – Difficulty of encoding:
a	U-CIT model
b	U-CIT entities
Q6	I found U-CIT useful.
Q7	I would use U-CIT in projects.
Q8	I would recommend U-CIT to others.
Q9	What was the most challenging part in the study?
Q10	Any suggestions to improve U-CIT?

constraints entered in the respective entity field and those in the model field were combined and fed to the constraint solver. The result was then displayed in the output frame, allowing the participant to check whether the entity can be covered in a valid test case. In both cases, when the constraints were satisfiable, the output frame presented a solution where each parameter defined in the U-CIT model took on a valid value. Otherwise, a warning message indicating that the constraints were not satisfiable, was emitted.

In addition to the Test buttons, we also had a "Generate" button, which fed all the constraints entered (the ones entered in the model and entity fields) to the cover-and-generate constructor (Chapter 5.1) to compute an U-CIT object. When an U-CIT object was created (which is, indeed, a set of test cases), it was displayed in the output frame in the form of a table, where rows represented the test cases generated and columns depicted the parameters defined in the U-CIT model (Figure 8.3). Furthermore, the entities covered by each test case are reported.

Table 8.4 Demographic information about the participants categorized based on their performances in addressing the second problem.

		degree level		knowledgeable of CIT		experience (in years) in programming		experience (in years) in software testing	
		cat.	count	cat.	count	cat.	count	cat.	count
successful formulations	after seeing only the description	undergrad.	2	yes	4	≤ 2	2	≤ 2	6
						3	1	3	0
		grad.	4	no	2	4	1	4	0
						≥ 5	2	≥ 5	0
	after seeing both the description and example	undergrad.	2	yes	1	≤ 2	0	≤ 2	4
						3	1	3	0
		grad.	2	no	3	4	2	4	0
						≥ 5	1	≥ 5	0
formulations with issues	missing constraints	undergrad.	0	yes	2	≤ 2	0	≤ 2	2
						3	1	3	0
		grad.	3	no	1	4	1	4	0
						≥ 5	1	≥ 5	1
	non-trivial generalization	undergrad.	3	yes	2	≤ 2	0	≤ 2	5
						3	2	3	0
		grad.	2	no	3	4	3	4	0
						≥ 5	0	≥ 5	0
give-ups		undergrad.	0	yes	2	≤ 2	0	≤ 2	1
						3	0	3	1
		grad.	2	no	0	4	0	4	0
						≥ 5	2	≥ 5	0

After completing all the studies, participants filled out an exit survey. Table 8.3 presents this survey.

8.2 Evaluation Framework

To evaluate the proposed approach, we first counted the number of successful formulations. For a given problem, we define a *successful formulation* as a formulation where both the U-CIT model and the entities to be covered are correctly expressed in a *generalizable manner*, such that an U-CIT object obtaining full coverage can be computed. Note that we also take the generalizability of the formulation into account because we observed that (solely for the second problem) some participants came up with formulations that are too specific for the problem instance given in the study and that, therefore, are non-trivial to generalize for other instances of the same problem. These formulations were often obtained by introducing additional constraints in the U-CIT model in an ad hoc manner just to avoid some undesirable results. More discussion on this can be found in Chapter 8.3.

We also measured the time it took for the participants to complete the study. More specifically, for a given problem, we measured the *completion time* as the difference between the time the description of the problem was presented to the participant and the time the participant completed the study. Note that a study was completed whenever the participant chose to finish the study. In all but two cases, this happened after computing an U-CIT object achieving full coverage. In two cases, however, the participants chose to stop working on the current problem in the middle of the study as they found the problem "very difficult" (see Chapter 8.3 for more information). Each participant worked on the problems one after another.

We, furthermore, counted the *number of errors* made by the participants. To this end, we counted the number of times the Test and the Generate buttons were clicked (Chapter 8.1) and the result obtained did not meet the expectation of the participant. More specifically, if a participant, after clicking on a Test button or a Generate button, made some changes and clicked on the same button, we assumed that the participant made an error before the first click (as the expectation of the participant after the first click did not seem to be met). Note that this metric provides an approximation of the number of errors made because on numerous occasions, we observed that the participants intentionally developed incorrect or missing constraints to test their hypotheses or to gain insight into the problem. We still opted to use this metric because attempting to figure out the actual intention of the participant after every click of a button would have introduced a great deal of intervention.

Furthermore, the percentage of the participants, who "agreed" or "strongly agreed" with a question group in the exit survey, was computed as the average percentage of the participants, who "agreed" or "strongly agreed" with the questions in the group. The percentage of the participants, who found the problems "difficult" or "very difficult," is computed in the same manner.

8.3 Data and Analysis

We first observed that all of the participants understood how U-CIT works. In particular, none of the participants were reminded of the basic U-CIT concepts during the study.

We then observed that the participants could also formulate previously unseen problems in U-CIT. More specifically, all of the participants successfully formulated the

first and the third problems. That is, for each of these problems, all of the participants correctly expressed the U-CIT model as well as all of the U-CIT entities to be covered in a generalizable manner and generated an U-CIT object obtaining full coverage. And, they did so in a relatively fast manner. The average time it took for the participants to complete these studies was 4.88 minutes ($min = 1.47$ and $max = 13.89$) for the first problem and 5.38 minutes ($min = 2.29$ and $max = 9.02$) for the third problem.

As expected, the participants found the second problem more difficult than the other two problems, which was also reflected on the outcomes of the exit survey. While 65% of the participants found the second problem "difficult" or "very difficult," none of the participants thought the same thing for the first and third problems. As a matter of the fact, based on the answers given to the open-ended Q9 (Table 8.3), the most challenging part in the entire study turned out to be expressing an U-CIT model for the second problem. Two participants, indeed, chose to terminate this study in the middle of it after spending 42.56 minutes on average as they found the problem "very difficult" (see the row marked with "give-ups" in Table 8.4 for the demographic information of these participants).

Half (10 out of 20) of the participants, however, successfully formulated the problem in a generalizable manner by using the same (or similar) approach introduced in Chapter 6.2.4 and obtained full coverage. 6 of them did so after seeing the description in Figure 8.1a and 4 after seeing both the description and the example flow problem in Figure 8.1b (see the rows marked with "after seeing only the description" and "after seeing both the description and example" in Table 8.4, respectively, for the demographic information of these participants). None of the participants, who came up with a generalizable solution for this problem, did so without seeing the description or the example. The average completion time was 47.14 minutes ($min = 25.79$ and $max = 69.49$).

The remaining 40% (8 out of 20) of the participants, although generated U-CIT objects obtaining full coverage, either came up with a formulation, the generalization of which was non-trivial, or covered some of the entities by chance. More specifically, 3 participants developed generalizable formulations by representing edges using Boolean variables (rather than using integer variables), which were quite similar to the formulation we developed in Chapter 6.2.4. However, the constraints, which should have invalidated the presence of multiple independent flows, were missing from the U-CIT models (see the row marked with "missing constraints" in Table 8.4 for the demographic information of these participants). The participants failed to identify the issue because their formulations happened to obtain full coverage by

generating valid test cases for the graph given in the study. Had they worked on larger graphs, however, they might have pinpointed and fixed the issue. The average completion time for this category of participants was 37.36 minutes ($min = 33.91$ and $max = 43.01$).

The remaining 5 participants developed formulations, the generalizations of which were non-trivial (see the row marked with "non-trivial generalization" in Table 8.4 for the demographic information of these participants). In particular, all of these participants chose to represent each vertex (rather than each edge) in the graph by using an integer variable, the value of which represents the order in which the vertex is visited. For a given variable, the set of possible values were determined manually by considering all possible paths that could be traversed. The invalid combinations of variable values (i.e., invalid paths) were then prevented by introducing model constraints in a rather ad hoc manner every time the participant observed that some of the generated test cases were invalid and/or some of the entities could not be covered. 4 (out of 5) of these participants correctly expressed all the constraints as well as the entities for the graph given in the study and obtained full coverage. The remaining participant, although had some missing and/or faulty constraints in the U-CIT model, happened to obtain full coverage by chance. The average completion time for this category of participants was 46.42 minutes ($min = 23.26$ and $max = 64.54$).

We did not find any correlations between the performances of the participants and their levels of degree, knowledge of CIT, or experiences in programming and testing. We, however, observed that the knowledge of the domain was influential in successfully completing the studies. More specifically, for the first and third problems, which required basic knowledge of programming and testing, all of the participants successfully formulated the problems in U-CIT. For the second problem, which required basic knowledge of network flow problems, all of the participants, who successfully formulated the problem in U-CIT, did so either after seeing a definition of the network flow problem or after seeing both the definition and an example flow problem. Table 8.4 provides demographic information about the participants categorized based on their performances in addressing the second problem.

Regarding the mistakes made during the study, we first observed that (as expected) the participants made more mistakes when solving the second problem, compared to the other two problems. While the average number of mistakes made for the second problem was 5.06, those for the first and third problems were 0.50 and 0.67, respectively.

We then observed a debugging pattern. The participants, solely for the second

Table 8.5 Responses to the exit survey given in Table 8.3.

	strongly disagree	disagree	neutral	agree	strongly agree
Q1a	0	0	0	2	18
Q1b	0	0	0	3	17
Q1c	0	0	0	4	16
Q2a	0	0	0	3	17
Q2b	0	0	0	6	14
Q2c	0	0	0	4	16
Q2d	0	0	0	7	13
Q6	0	0	0	7	13
Q7	0	0	0	7	13
Q8	0	0	0	4	16
	very difficult	difficult	normal	easy	very easy
Q3a	0	0	1	3	16
Q3b	0	0	1	4	15
Q4a	1	12	7	0	0
Q4b	1	6	9	4	0
Q5a	0	0	0	2	18
Q5b	0	0	0	2	18

problem, found expressing the entities as constraints easier than expressing the U-CIT models. They, therefore, used the entity constraints to debug the models. More specifically, to gain insight as well as to test their hypotheses, they tended to click on the Test buttons associated with the entity fields. When the results obtained were not expected, they modified and fixed the models.

Another interesting observation we made is that more than half of the participants (especially for the first and third problems) encoded the U-CIT models as they were reading the study descriptions. That is, as they discovered new system constraints (e.g., parameters and their domains), they updated the models, suggesting that they knew what to look for in the requirements to develop the U-CIT models.

Last but not least, Table 8.5 presents the outcome of the exit survey. Regarding the questions Q1-Q2 (i.e., Q1.a-Q1.c and Q2.a-Q2.d) and Q6-Q8, all of the participants "agreed" or "strongly agreed" that 1) they understood the basic concepts both in constraint solving and in U-CIT, 2) they found U-CIT useful, 3) they would use U-CIT in a project, and 4) they would recommend U-CIT to others.

Regarding Q10, one suggestion was to improve the syntax of the language we used for expressing the constraints in a way that closely resembles the Boolean expressions used in main stream programming languages. Another suggestion was to develop a means of expressing "long and repetitive" constraints in a more efficient and effective manner, which, in turn, can further simplify the processes of developing the constraints.

8.4 Discussion

We observed that some problems are more difficult to formulate in U-CIT than others. This is, indeed, to be expected. After all, solving some problems may require specific background knowledge and not everybody may possess it. Note, however, that the proposed approach still allows experts to formulate such problems in U-CIT and others to use the existing formulations to compute U-CIT objects. For example, the U-CIT formulation we developed to express reachability in graph-based models, can be used to obtain full coverage under other reachability-based coverage criteria (other than the ones we studied in this work) by changing the entities to be covered.

9. GENERAL DISCUSSION

In this chapter, we informally discuss the proposed approach in an attempt to 1) address the additional questions that the reader may have, 2) discuss the big picture, in which we envision U-CIT to be an integral part, and 3) present possible avenues for future research.

U-CIT is not a methodology for choosing the entities to be tested. It rather takes as input a set of testable entities and aims to find a minimum number of test cases, such that every required entity is covered by at least one test case.

In the absence of a methodology or a tool that can automatically determine what needs to be tested, such as the existing structural code coverage criterion we used in Chapter 6.1, identifying the set of entities to be tested may require some effort. Note, however, that if the entities at question should really be tested, then they, in one way or another, must be defined and enumerated regardless of whether U-CIT or a specialized constructor is used.

Once the entities are determined, one may consider developing a specialized CIT constructor. To do that, however, a procedure, which determines whether a given set of entities can be covered together in a test case or not, needs to be devised. But, then, the very same procedure can be used as the "solver" in U-CIT, which in turn offers a constructor for free. Note that, as we have discussed in Chapter 5.5, given such a procedure, the entities can be represented in any form desired (e.g., not necessarily in formal logic), since U-CIT does not need to interpret them.

After all, developing specialized CIT constructors may not be easy. As a matter of fact, we introduced our generate-and-cover constructor (Algorithm 2) to mimic one of the simplest ways of generating CIT objects: Keep on randomly generating valid test cases until all the required entities have been covered. However, developing a high-performing specialized constructor is quite challenging, which is also apparent from more than 50 papers published in the literature, the sole purpose of which is to compute standard covering arrays Nie & Leung (2011). Therefore, our goal is to generalize the construction as much as possible, so that the collective effort

spent for developing U-CIT constructors can be leveraged in a wider spectrum of test scenarios, which in turn increases the flexibility of CIT.

10. THREATS TO VALIDITY

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our studies to industrial practice.

One threat concerns the representativeness of the case studies as well as the subject applications used in the experiments. To alleviate this issue, we addressed a different CIT problem in each case study.

In the first study (Chapter 6.1), we enhanced standard CIT with a well-known structural code coverage metric, namely decision coverage, and conducted comparative studies on 12 well-known software systems, including `Python`, `vim`, and `xterm`. In the second study (Chapter 6.2), we enhanced a number of existing order-based coverage criteria. In the third study (Chapter 6.3), we developed solutions for a problem faced by a successful consumer electronics company and evaluated the proposed approach by using the data collected from the field. Furthermore, not only the CIT problems we have addressed, but also the solution approaches we have developed were diverse. In the first study (Chapter 6.1), the values of the parameters were Boolean constraints, rather than discrete values, and we used U-CIT with a SAT solver. In the second study (Chapter 6.2), we expressed the reachability problem in DAGs as a constraint satisfaction problem (CSP) and used a CSP solver. In the third study (Chapter 6.3), we worked with parameters, each of which takes on a value from a discrete set of values and used U-CIT with a simple, application-specific constraint solver.

We have, however, not directly studied the fault-detection abilities of the U-CIT objects we computed. In the first study (Chapter 6.1), we developed U-CIT objects to obtain full decision coverage. The decision coverage criterion is, indeed, a well-known structural code coverage criterion for measuring the adequacy of test suites. Therefore, its fault-detection abilities are well-studied (Cai & Lyu, 2005). In the second study (Chapter 6.2), we enhanced a number of existing order-based coverage criteria, which have already been shown to be effective in testing event-

driven systems (Yuan et al., 2011). In the third study (Chapter 6.3), we developed usage-based CIT objects to reduce the size of the interaction test suites by covering only the t-tuples (or a fraction of them) seen in the field. Consequently, for the test scenarios, in which standard covering arrays are infeasible (or undesirable) due to their sizes, usage-based CIT objects would offer the same (or similar) fault revealing abilities for the faults caused by the t-tuples seen in the field.

The number of times we repeated the experiments in the paper varied depending on the cost of the respective experiments. We, however, opted to work on larger CIT problems with smaller repetition counts, rather than working on smaller formulations with larger repetition counts. Furthermore, for each study, we have added a discussion for the costs involved in the study. The actual costs, however, may vary depending on the experience of the tester.

For the hints approach, the testable entities needs to be defined as conjunction of smaller parts. However, it may not always be straight forward to represent the testable entities as conjunction of smaller parts. In such cases, even though, the hints approach would not work, the standard U-CIT will still be a useful tool to compute the requested U-CIT object.

Regarding the user studies, all the participants in these studies were students. We, however, had both undergraduate- and graduate-level students with some background on software testing. Furthermore, more than half of these students had taken at least one course where standard CIT approaches were studied. A related threat concerns the representativeness of the problems used in the user studies. We, however, used the smaller instances of the very same problems we studied in this work (Chapters 6.1-6.3). Furthermore, these problems were not known to the participants before taking part in the study. The participants were asked to finish working on one problem before moving to the subsequent problem. They were also required to finish all the studies in a single session. Had they been given more time and/or more instances of the same problems, more participants might have successfully formulated them in a generalizable manner and/or identified and fixed the issues with their formulations.

11. CONCLUDING REMARKS

In this work, we have first presented U-CIT to make combinatorial interaction testing more flexible. In U-CIT, both the testable entities to be covered and the space of test cases, from which the samples will be drawn, are expressed as constraints. Consequently, the problem of computing U-CIT objects, turns into an interesting constraint solving problem, which we call *cov-CSP*. Given a set of constraints, each representing a testable entity to be covered, cov-CSP aims to divide the set into a minimum number of satisfiable clusters, such that a solution for a cluster represents a test case, covering the testable entities included in the cluster. The collection of all the test cases computed for the clusters constitute the U-CIT object, covering each required testable entity at least once.

We have then developed two constructors, namely cover-and-generate and generate-and-cover, to solve the cov-CSP problem, thus to compute U-CIT objects. These constructors can work with any types of constraints as long as an appropriate solver, the purpose of which is to determine whether a given set of entities can be covered in a single test case or not, is provided.

To evaluate U-CIT, we have first carried out three case studies, each of which focused on a different CIT problem, demonstrating that U-CIT is more flexible than the existing CIT approaches. We have arrived at this conclusion by noting that, in these studies, it was either unclear how to use the existing constructors (if at all possible) to compute the requested CIT objects, or the existing constructors required non-trivial modifications or excessive number of test cases to guarantee full coverage. U-CIT, on the other hand, used the same U-CIT constructor to compute all the requested CIT objects these studies without requiring any modifications.

Moreover, we empirically demonstrated that the performance (i.e., the construction times) of U-CIT constructor can be significantly improved by using "hints". Hints can improve the efficiency of U-CIT constructors by reducing the number of times the solver is called and/or by calling the solver with simpler constraints.

Last but bot least, we have also carried out user studies to further evaluate U-

CIT, demonstrating the usability of the proposed approach. One thing we observed in these studies is that some problems are more difficult to formulate in U-CIT than others as they require some specific background knowledge, which may not be possessed by everybody.

A possible future work, tools (e.g., front-ends) that can provide various means for defining the coverage criteria as well as the input spaces, such that the testable entities selected by the coverage criteria are automatically generated, can be developed to improve the usability of U-CIT in the field. Another avenue for future work is to develop alternative approaches for solving the cov-CSP problem, i.e., developing better constructors. We also plan to investigate the fault revealing capabilities of U-CIT. Lastly, the computational complexity analysis of the problems and the asymptotic time complexity of the algorithms are missing and left for future work.

BIBLIOGRAPHY

Ahlswede, R., Cai, N., Li, S.-Y., & Yeung, R. W. (2000). Network information flow. *IEEE Transactions on information theory*, *46*(4), 1204–1216.

Akhtar, Y. & Maity, S. e. a. (2016). Generating test suites with high 3-way coverage for software testing. In *2016 IEEE International Conference on Computer and Information Technology (CIT)*, (pp. 10–17).

Alazzawi, A. K. & Rais, H. M. e. a. (2019). Phabc: A hybrid artificial bee colony strategy for pairwise test suite generation with constraints support. In *2019 IEEE Student Conference on Research and Development (SCOReD)*, (pp. 106–111).

Apache Software Foundation (2014). Apache ActiveMQ.

Banbara, M. & Inoue, K. e. a. (2017). catnap: Generating test suites of constrained combinatorial testing with answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, (pp. 265–278). Springer.

Banbara, M., Matsunaka, H., Tamura, N., & Inoue, K. (2010). Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, (pp. 112–126). Springer.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press.

Bazaraa, M. S., Jarvis, J. J., & Sherali, H. D. (2011). *Linear programming and network flows*. John Wiley & Sons.

Belli, F., Beyazit, M., & Güler, N. (2011). Event-Based GUI testing and reliability assessment techniques–an experimental insight and preliminary results. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 212–221). IEEE.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., Zhu, Y., & et al. (2003). Bounded model checking. *Advances in computers*, *58*(11), 117–148.

Biere, A., Heule, M., & van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

Blue, D., Hicks, A., Rawlins, R., & Tzoref-Brill, R. (2019). Practical fault localization with combinatorial test design. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 268–271).

Bombarda, A. & Gargantini, A. (2020). An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 157–166).

Bruening, D. L. (1999). *Systematic testing of multithreaded Java programs*. PhD thesis, Massachusetts Institute of Technology.

Bryce, R. C. & Colbourn, C. J. (2006). Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, *48*(10), 960 – 970. Advances in Model-based Testing.

Cai, X. & Lyu, M. R. (2005). The effect of code coverage on fault detection under

different testing profiles. *ACM SIGSOFT software engineering notes*, *30*(4), 1–7.

Çalpur, M. Ç. (2012). Interleaving coverage criteria oriented testing of multithreaded applications. Master's thesis.

Charbachi, P., Eklund, L., & Enoiu, E. (2017). Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers? In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, (pp. 92–99).

Chee, Y. M., Colbourn, C. J., Horsley, D., & Zhou, J. (2013). Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, *27*(4), 1844–1861.

Chilenski, J. J. & Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, *9*(5), 193–200.

Choi, J. S. (2017). Controller-centric combinatorial wrap-around interaction testing to evaluate a stateful pce-based transport network architecture. *IEEE/OSA Journal of Optical Communications and Networking*, *9*(9), 792–802.

Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. on Soft. Eng.*, *23*(7), 437–44.

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

Cohen, M. B., Dwyer, M. B., & Shi, J. (2008). Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, *34*(5), 633–650.

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

Czerwonka, J. (2008). Pairwise testing in the real world: Practical extensions to test-case scenarios. *Microsoft Corporation, Software Testing Technical Articles*.

de Moura, L. M. & Bjørner, N. (2009). Satisfiability modulo theories: An appetizer. *SBMF*, *5902*, 23–36.

Demiroz, G. & Yilmaz, C. (2012). Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles*, (pp. 9–16).

Deng, X., Zhang, Z., Li, R., Yan, J., & Zhang, J. (2020). Combinatorial testing of browsers' support for multimedia. *IEEE Transactions on Reliability*, *69*(4), 1323–1340.

Dumlu, E., Yilmaz, C., Cohen, M. B., & Porter, A. (2011). Feedback driven adaptive combinatorial testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, (pp. 243–253). ACM.

Eén, N. & Sörensson, N. (2003). An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, (pp. 502–518). Springer.

Eiter, T., Ianni, G., & Krennwallner, T. (2009). Answer set programming: A primer. In *Reasoning Web International Summer School*, (pp. 40–110). Springer.

Eitner, C. & Wotawa, F. (2019). Crucial tool features for successful combinatorial input parameter testing in an industrial application. In *2019 IEEE Interna-*

tional Conference on Software Testing, Verification and Validation Workshops (ICSTW), (pp. 188–189).

Fouché, S., Cohen, M. B., & Porter, A. (2009). Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, (pp. 177–188). ACM.

Galinier, P., Kpodjedo, S., & Antoniol, G. (2017). A penalty-based Tabu search for constrained covering arrays. In *Proceedings of the Genetic and Evolutionary Computation Conference*, (pp. 1288–1294). ACM.

Gao, F., Deng, F., & Yan, Y. (2019). Research on multi-constraint combinatorial test technology for high confidence embedded software (s). In *SEKE*, (pp. 102–140).

Garvin, B. J., Cohen, M. B., & Dwyer, M. B. (2011). Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, *16*(1), 61–102.

Gladisch, C., Heinzemann, C., Herrmann, M., & Woehrle, M. (2020). Leveraging combinatorial testing for safety-critical computer vision datasets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, (pp. 324–325).

Hnich, B., Prestwich, S., & Selensky, E. (2004). Constraint-based approaches to the covering test problem. In *International Workshop on Constraint Solving and Constraint Logic Programming*, (pp. 172–186). Springer.

Hnich, B., Prestwich, S. D., Selensky, E., & Smith, B. M. (2006). Constraint models for the covering test problem. *Constraints*, *11*(2-3), 199–219.

Hopcroft, J. E. (2013). *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Pearson Education India.

Jarman, D. & Smith, R. e. a. (2019). Applying combinatorial testing to large-scale data processing at adobe. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 190–193). IEEE.

Javeed, A. (2015). Gray-box combinatorial interaction testing. Master's thesis.

Javeed, A. & Yilmaz, C. (2015). Combinatorial interaction testing of tangled configuration options. In *2015 IEEE Eight International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 1–4). IEEE.

Jenkins, B. (2005). jenny: A pairwise testing tool. `http://www.burtleburtle.net/bob/index.html`.

Jin, H., Kitamura, T., Choi, E.-H., & Tsuchiya, T. (2018). A satisfiability-based approach to generation of constrained locating arrays. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 285–294). IEEE.

Johansen, M. F., Haugen, Ø., & Fleurey, F. (2012). An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, (pp. 46–55). ACM.

Katebi, H., Sakallah, K. A., & Marques-Silva, J. P. (2011). Empirical study of the anatomy of modern sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, (pp. 343–356). Springer.

Klammer, C., Ramler, R., & Stummer, H. (2016). Harnessing automated test case generators for gui testing in industry. In *2016 42th Euromicro Conference on*

Software Engineering and Advanced Applications (SEAA), (pp. 227–234).

Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., & Lei, Y. (2012a). Combinatorial methods for event sequence testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, (pp. 601–609). IEEE.

Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., & Lei, Y. (2012b). Combinatorial methods for event sequence testing. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, (pp. 601–609).

Kuhn, D. R., Kacker, R. N., Lei, Y., et al. (2010). Practical combinatorial testing. *NIST special Publication*, *800*(142), 142.

Lacchia, M. (2018). Radon.

Lawrence, J., Kacker, R. N., Lei, Y., Kuhn, D. R., & Forbes, M. (2011). A survey of binary covering arrays. *the electronic journal of combinatorics*, *18*(1), P84.

Lei, Y., Carver, R. H., Kacker, R., & Kung, D. (2007). A combinatorial testing strategy for concurrent programs. *Software Testing, Verification and Reliability*, *17*(4), 207–225.

Li, Z., Chen, Y., & Gong, e. a. (2019). A survey of the application of combinatorial testing. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, (pp. 512–513).

Liebig, J., Apel, S., Lengauer, C., Kästner, C., & Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, (pp. 105–114). ACM.

Lin, J. & Luo, C. e. a. (2015). Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, (pp. 494–505). IEEE.

Lombardy, S., Régis-Gianas, Y., & Sakarovitch, J. (2004). Introducing vaucanson. *Theoretical Computer Science*, *328*(1-2), 77–96.

Lopez-Herrejon, R. E., Fischer, S., Ramler, R., & Egyed, A. (2015). A first systematic mapping study on combinatorial interaction testing for software product lines. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 1–10).

Lu, S., Jiang, W., & Zhou, Y. (2007). A study of interleaving coverage criteria. In *the 6th joint meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, (pp. 533–536). ACM.

Luo, C. & Lin, J. e. a. (2021). Autoccag: An automated approach to constrained covering array generation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, (pp. 201–212).

László, F. T. (2018). SATisPy. `https://github.com/netom/satispy`.

Ma, L., Zhang, F., Xue, M., Li, B., Liu, Y., Zhao, J., & Wang, Y. (2018). Combinatorial testing for deep learning systems. *arXiv preprint arXiv:1806.07723*.

Makaś, M. (2016). A certain version of preservationism. *Logic and Logical Philosophy*, *26*(1), 63–77.

Marek, V. W. & Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm* (pp. 375–398).

Springer.

Mats, G., Jeff, O., & Jonas, M. (2006). Handling constraints in the input space when using combination strategies for software testing. Technical Report HS-IKI -TR-06-001, University of Skövde, School of Humanities and Informatics.

Memon, A. M. (2007). An event-flow model of GUI-based applications for testing. *Software testing, verification and reliability*, *17*(3), 137–157.

Mercan, H. & Yilmaz, C. (2016). A constraint solving problem towards unified combinatorial interaction testing. In *Proceedings of the 7th Workshop on Constraint Solvers in Testing, Verification, and Analysis*, volume 1639, (pp. 24–30). CEUR.

Mercan, H. & Yilmaz, C. (2021). Computing sequence covering arrays using unified combinatorial interaction testing. In *Proceedings of the 10th International Workshop on Combinatorial Testing (accepted, to be published)*.

Mercan, H., Yilmaz, C., & Kaya, K. (2018). Chip: A configurable hybrid parallel covering array constructor. *IEEE Transactions on Software Engineering*, *45*(12), 1270–1291.

Michaels, R., Adamo, D., & Bryce, R. (2020). Combinatorial-based event sequences for reduction of android test suites. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, (pp. 0598–0605).

Mirzaei, N. & Garcia, J. e. a. (2016). Reducing combinatorics in gui testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, (pp. 559–570).

Mohammad, S. A. K. & Valepe, S. V. e. a. (2019). A comparative study of the effectiveness of meta-heuristic techniques in pairwise testing. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, (pp. 91–96).

Mukelabai, M. & Nešić, D. e. a. (2018). Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 155–166).

Musuvathi, M., Qadeer, S., Ball, T., Musuvathi, M., Qadeer, S., & Ball, T. (2007). Chess: A systematic testing tool for concurrent software. Technical report, Technical Report MSR-TR-2007-149, Microsoft Research.

Nie, C. & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, *43*(2), 11.

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, *25*(3-4), 241–273.

Perron, L. & Furnon, V. (2019). OR-Tools.

Petke, J., Cohen, M. B., Harman, M., & Yoo, S. (2015). Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, *41*(9), 901–924.

Qi, G., Tsai, W.-T., Colbourn, C. J., Luo, J., & Zhu, Z. (2018). Test-algebra-based fault location analysis for the concurrent combinatorial testing. *IEEE Transactions on Reliability*, *67*(3), 802–831.

Qian, Y., Zhang, C., & Wang, F. (2018). Selecting products for high-strength t-wise testing of software product line by multi-objective method. In *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*,

(pp. 370–378).

Rao, C. & Li, N. e. a. (2021). Combinatorial test generation for multiple input models with shared parameters. *IEEE Transactions on Software Engineering*, 1–1.

Rescher, N. & Manor, R. (1970). On inference from inconsistent premisses. *Theory and decision*, *1*(2), 179–217.

Samuel, P. & Joseph, A. T. (2008). Test sequence generation from uml sequence diagrams. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08*, (pp. 879–887). IEEE.

Schotch, P. K. & Jennings, R. E. (1980). Inference and necessity. *Journal of Philosophical Logic*, *9*(3), 327–340.

Schroeder, P. J., Faherty, P., & Korel, B. (2002). Generating expected results for automated black-box testing. In *In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002*, (pp. 139–148). IEEE.

Sheng, Y., Jiang, S., & Wei, C. (2019). Constructing test suites for real-time embedded systems under input timing constraints. *IEEE Access*, *7*, 20920–20937.

Tamura, N. & Banbara, M. (2008). Sugar: A csp to sat translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, 65–69.

Williams, A. W. & Probert, R. L. (1996). A practical strategy for testing pairwise coverage of network interfaces. In *Proceedings of Seventh International Symposium on Software Reliability Engineering*, (pp. 246–254). IEEE.

Wu, H., Changhai, N., Petke, J., Jia, Y., & Harman, M. (2019). Comparative analysis of constraint handling techniques for constrained combinatorial testing. *IEEE Transactions on Software Engineering*, 1–1.

Wu, H., Nie, C., Kuo, F.-C., Leung, H., & Colbourn, C. J. (2014). A discrete particle swarm optimization for covering array generation. *IEEE Transactions on Evolutionary Computation*, *19*(4), 575–591.

Yan, J. & Zhang, J. (2006). Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *Computer Software and Applications Conference. 30th Annual International*, volume 1, (pp. 385–394). IEEE.

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

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

Yilmaz, C., Cohen, M. B., & Porter, A. A. (2006a). Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng.*, *32*(1), 20–34.

Yilmaz, C., Cohen, M. B., & Porter, A. e. a. (2006b). Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, *32*(1), 20–34.

Yilmaz, C., Dumlu, E., Cohen, M. B., & Porter, A. (2014). Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. *IEEE Transactions on Software Engineering*, *40*(1), 43–66.

Yilmaz, C., Fouche, S., Cohen, M. B., Porter, A., Demiroz, G., & Koc, U. (2014). Moving forward with combinatorial interaction testing. *Computer*, *47*(2), 37–

45.

Yu, L., Lei, Y., Kacker, R. N., & Kuhn, D. R. (2013). ACTS: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 370–375). IEEE.

Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R. N., & Kuhn, D. R. (2013). An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, (pp. 242–251). IEEE.

Yu, Y. T. & Lau, M. F. (2006). A comparison of mc/dc, mumcut and several other coverage criteria for logical decisions. *Journal of Systems and Software*, *79*(5), 577–590.

Yuan, X., Cohen, M. B., & Memon, A. M. (2011). GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, *37*(4), 559–574.

Zhang, Y., Cai, L., & Ji, W. (2017). Combinatorial testing data generation based on bird swarm algorithm. In *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, (pp. 491–499).