Constrained Locating Arrays and Constrained Detecting Arrays: New Mathematical Structures for Fault Identification in Combinatorial Testing

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2022

Hao JIN

ABSTRACT

Combinatorial interaction testing (CIT) is an efficient black-box software testing method that can detect failures triggered by interactions of features or components in software systems. Existing research has shown that most such interactions only involve a small number of features or components; hence, it is sufficient to test all interactions among two to six test parameters to reveal system faults. Covering arrays (CAs) are mathematical structures designed to test all interactions among a fixed number (e.g., two or three) of parameters. By avoiding testing all interactions with all parameters, CAs can significantly reduce the testing cost.

However, it is not always possible to use CAs directly in practical testing. In most cases, real-world systems have complex requirements and restrictions among their testing space. To test software systems correctly, all test cases in a test suite must satisfy the constraints imposed by these requirements and restrictions. To overcome this weakness, constrained CAs (CCAs) which are CAs that only contain constraint-satisfying test cases, have been used. Because some interactions may violate constraints as well, CCAs only require testing all interactions that satisfy constraints.

Although CCAs are able to detect the existence of faulty interactions in systems with constraints at low cost, CCAs cannot identify them from test outcomes. This is because CCAs do not have a sufficient number of test cases to distinguish faulty from non-faulty interactions.

With the aim of not only detecting the existence of faulty interactions but also identifying them, locating arrays (LAs) and detecting arrays (DAs) have previously been proposed. LAs and DAs are mathematical structures that can also be used as test suites in CIT. Although they are slightly larger than CAs, LAs and DAs can identify faulty interactions from their test outcomes. In other words, both LAs and DAs preserve the benefit of low testing cost from CAs while also being able to identify faulty interactions. Similar to CAs, LAs and DAs do not take constraints into account. Thus, they cannot be used for testing of systems with constraints.

This dissertation proposes the notions of constrained LAs (CLAs) and constrained DAs (CDAs). These arrays are mathematical structures that can be used for fault identification in the presence of constraints. CLAs and CDAs extend LAs and DAs to testing systems that have constraints on the test space. In short, CLAs and CDAs require that all test cases must satisfy constraints. Thus, CLAs and CDAs enhance the applicability of LAs and DAs to practical testing problems. In this dissertation, the mathematical properties of CLAs and CDAs are investigated.

This dissertation also proposes two algorithms for constructing both CLAs and CDAs. One algorithm is able to generate minimum CLAs and CDAs. It translates the problem of generating arrays into the satisfiability problem for logical expressions. Using an off-the-shelf satisfiability checker, the algorithm is able to generate CLAs and CDAs with minimum sizes. In contrast, the other algorithm is a heuristic algorithm that can generate minimum or near-minimum CLAs and CDAs in a short time.

Three experiments were conducted in the dissertation. The results of the first experiment show that the satisfiability-based generation algorithm can always generate minimum arrays given sufficient generation time. The results of the second experiment show that the heuristic generation algorithm scales to problems of practical sizes. In the third experiment, CLAs and CDAs were applied to two real-world applications. The experi-

mental results show that faulty interactions were successfully identified using CLAs and CDAs.

Both CLAs and CDAs provide new choices to testers of software systems. The cost-efficiency of CLAs and CDAs is appealing, especially for the testing of large-scale systems, such as highly configurable systems.

LIST OF MAJOR PUBLICATIONS

(1) Hao Jin, Takashi Kitamura, Eun-Hye Choi, and Tatsuhiro Tsuchiya, A satisfiability-based approach to generation of constrained locating arrays, In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW2018)*, pp. 285–294, April 2018.

(2) Hao Jin and Tatsuhiro Tsuchiya, Deriving fault locating test cases from constrained covering arrays, In *Proceedings of 2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC2018)*, pp. 233–240, December 2018.

(3) Hao Jin, Ce Shi, and Tatsuhiro Tsuchiya, Constrained detecting arrays for fault localization in combinatorial testing, In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC'20)*, pp. 1971–1978, March 2020.

(4) Hao Jin and Tatsuhiro Tsuchiya, A two-step heuristic algorithm for generating constrained detecting arrays for combinatorial interaction testing, In *Proceedings of 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE2020)*, pp. 219–224, September 2020.

(5) Hao Jin and Tatsuhiro Tsuchiya, Constrained locating arrays for combinatorial interaction testing, *Journal of Systems and Software*, vol. 170, no. 110771, December 2020.

(6) Hao Jin, Ce Shi, and Tatsuhiro Tsuchiya, Constrained detecting arrays: mathematical structures for fault identification in combinatorial interaction testing (submitted).

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

Software systems play a core role in society. With the wide spread of software system usage, the dependability of software systems is a vital issue to address. Although many methods have been proposed to ensure the correctness of software systems, *software testing* is still a central method for guaranteeing that software systems execute as planned. Software testing uses pre-defined input to examine software systems. By comparing the output of the systems with expected results, the latent faults can be found and then corrected. If one were able to enumerate all possible inputs and expected outputs of a system, it would be not difficult to use them in a test plan. However, this strategy, often called *exhaustive testing*, is usually impractical for most systems because its cost increases exponentially with the scale of the system, and the goal of software testing is to find more faults with less cost.

1.2 Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) is an efficient testing strategy that aims to test all interactions among a specified number of system parameters or system components. By avoiding testing all interactions with all parameters, CIT test suites can significantly reduce the cost of testing. The reasons for choosing CIT for testing are reported in [34, 35, 37]. The research shows that it is sufficient to only test interactions involving a small number of parameters to reveal most of the latent faults. Checking the interactions among two to six parameters is generally sufficient to ensure that a system works correctly. Unlike with exhaustive testing, the testing cost of CIT grows logarithmically as the number of system parameters increases. Using CIT for software development can reduce testing cost significantly. Surveys on CIT research can be found in, for example, [9, 22, 46, 66].

CIT is based on mathematical structures in the field of combinatorial designs. The most typical class of mathematical structures used for CIT is t-way *covering arrays* (t-CAs). In a t-CA, every interaction involving t parameters appears in at least one test case; thus, the use of a t-CA ensures that all t-way interactions are exercised. In other words, all faults that are caused by t-way interactions will be detected by t-CAs.

There are many directions to expand the capability of CIT. One is to add fault localization capability to test suites. The (d, t)-*locating arrays* (LAs) and (d, t)-*detecting arrays* (DAs) proposed in [10] represent test suites that can not only detect but also identify faulty interactions. The integers d and t are predefined parameters, where d represents the number of faulty interactions that can be identified and t represents the number of parameters involved in the faulty interactions. LAs and DAs add this capability to CAs at the cost of an increased number of test cases. One of the differences between (d, t)-LAs and (d, t)-DAs appears when there are more than d faulty interactions. In such a case,

(d, t)-LAs may incorrectly regard faulty interactions as non-faulty because of the lack of test cases. However, (d, t)-DAs can indicate when the number of faulty interactions exceeds the ability of DAs for the same case and will never identify faulty interactions as non-faulty. The other difference is that LAs usually have less test cases, i.e., lower testing cost, than DAs when d and t are the same.

Another direction for expanding CIT is to incorporate constraints. Real-world systems usually have constraints on the input space. These constraints originate from, for example, user-defined requirements or running environment restrictions. To test systems with constraints correctly, proper handling of the constraints is necessary. For example, all test cases must satisfy the constraints.

In addition, constraints may make some interactions no longer testable. These *invalid* interactions require additional handling. *Constrained CAs (CCAs)* are an extension of CAs in which such constraints are incorporated. Many previous studies on CIT have tackled the problem of generating CCAs of small sizes [38, 40, 59].

1.3 Contribution of This Dissertation

The purpose of this dissertation is to extend the notion of LAs and DAs to widen their applicability to practical testing problems. In this dissertation, *constrained LAs* (CLAs) and *constrained DAs* (CDAs) are proposed. CLAs and CDAs are mathematical structures that can be used as test suites for identifying faulty interactions in the presence of constraints. CLAs and CDAs retain the properties of LAs and DAs while enabling fault identification for systems with constraints. CLAs have less testing cost when compared to CDAs. However, CDAs can identify faulty interactions more accurately.

In addition to the presence of invalid interactions, constraints may cause some valid interactions to become no longer identifiable. That is, constraints may make it impossi-

ble to identify a non-faulty interaction or set of such interactions from faulty interactions; hence, a special treatment is required to deal with such an inherently non-identifiable pair. To address this problem, the concepts of *distinguishability* and *masking* are proposed. The mathematical properties of the newly proposed structures, CLAs and CDAs, are investigated. Also, two generation methods are proposed in this dissertation for both CLAs and CDAs. One generation method can construct minimum arrays given sufficient time, while the other can construct minimum or near-minimum arrays in a short time.

1.4 Overview of Dissertation

This dissertation is organized as follows:

Chapter 2 introduces basic notions in the field of CIT, including system models, test cases, test suites, and interactions. Then, the definitions of existing mathematical structures, such as CAs, LAs, and DAs, are introduced. CCAs, the constrained version of CAs, are then defined. Examples of these arrays are also presented in this chapter.

Chapter 3 introduces CLAs, the constrained version of LAs. To propose CLAs, the notion of distinguishability, which is a relation that constraints may induce on interactions, is clarified. Then, the definition of LAs is relaxed so that the newly defined CLAs can be applied to systems with constraints. Examples of CLAs are presented in this chapter, and the properties of CLAs are also discussed.

Chapter 4 introduces CDAs, the constrained version of DAs. The notion of masking, which is caused by constraints, is clarified. Using this notion, CDAs are then defined. The properties of CDAs are also discussed in this chapter.

Chapter 5 demonstrates how CLAs and CDAs can be used in fault identification processes.

Chapter 6 proposes generation methods for CLAs and CDAs. Two generation methods

are designed for each type of array; one method aims to generate minimum arrays, and the other generates minimum or near-minimum arrays in a short time.

Chapter 7 evaluates the fault identification capabilities of CLAs and CDAs and their generation methods. An evaluation compares the two methods in terms of generation time and generated array size. Then, an evaluation on generating arrays with different strengths t of interactions is performed. Finally, the performance of fault identification using CLAs and CDAs is examined in open source applications.

Chapter 8 summarizes related work, and Chapter 9 concludes this dissertation.

CHAPTER 2

PRELIMINARIES

2.1 Basic Notions of CIT

2.1.1 System Under Test

A *system under test* (SUT) is modeled as a 3-tuple $\mathcal{M} = \langle \mathcal{F}, \mathcal{S}, \phi \rangle$. $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$ is a set of parameters in the system, where k is the number of all parameters. $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ is a set of domains for all the parameters in $\mathcal{F}$, where S_i is the domain for the parameter F_i. Each domain S_i consists of two or more integers ranging from 0 to $|S_i| - 1$, i.e., $S_i = \{0, 1, \ldots, |S_i| - 1\}$. Different integers in S_i represent different values for the parameter F_i. $\phi : S_1 \times S_2 \times \cdots \times S_k \rightarrow \{true, false\}$ is a mapping representing the system constraints. Parameters and their domains can be regarded as the input of a program or as system configurations.

Table 2.1 shows a running example of a simple online shopping application SUT. There is a set of 4 parameters that take 3, 2, 3, and 4 different values in the system. The constraints in the SUT originate from real-world restrictions or requirements. This example is taken from [25]. The SUT originally included only one constraint ϕ_1. The constraint ϕ_2 is newly added to the original example for demonstration purposes.

Table 2.1: SUT: an online shopping mobile application [25]

$\mathcal{F}$	F_1 (Total Price)	F_2 (Shipping Address)	F_3 (Shipping Method)	F_4 (Payment Method)
$\mathcal{S}$	0: \$50	0: Domestic	0: Same-Day Delivery	0: Visa
	1: \$500	1: International	1: 2-Day Delivery	1: Mastercard
	2: \$1000	– –	2: 7-Day Delivery	2: Paypal
	– –	– –	– –	3: Gift Card
ϕ	ϕ_1 : Shipping Address = *International* ⇒ Shipping Method ≠ *Same-Day Delivery*			
	ϕ_2 : Payment = *Gift Card* ⇒ Shipping Address = *Domestic* ∧ Shipping Method = *Same-Day Delivery*			

2.1.2 Test Cases, Test Suites, and Interactions

A *test case* is an element of $S_1 \times S_2 \times \cdots \times S_k$; that is, a test case is a k-tuple $\langle \sigma_1, \ldots, \sigma_i, \ldots, \sigma_k \rangle$ such that $\sigma_i \in S_i$. A *test suite* is a set of test cases. A test suite is regarded as an $N \times k$ *array*, where each row represents a test case and there are N test cases. The *size* of a test suite (i.e., array) is the number of test cases (rows) in it. Hereinafter, the terms test suite and array are used interchangeably.

An *interaction* is a set of parameter-value pairs such that no parameters are overlapped. The *strength* of an interaction is the number of parameter-value pairs in the interaction. That is, $\{(F_{i_1}, \sigma_1), \ldots, (F_{i_t}, \sigma_t)\}$ is an interaction of strength t if and only if (iff) $F_{i_j} \neq F_{i_k}$ for any i_j, i_k $(i_j \neq i_k)$ and $\sigma_j \in S_{i_j}$ for all $i_j \in \{i_1, \ldots, i_t\}$. In this dissertation, λ is used instead of $\emptyset$ to denote the interaction of strength 0, i.e., the empty set. An interaction is called t-way iff its strength is t. An interaction T_1 *covers* another interaction T_2 iff $T_2 \subseteq T_1$. A set of interactions $\mathcal{T}$ are *independent* iff $T_1 \not\subseteq T_2$ for any $T_1, T_2 \in \mathcal{T}, T_1 \neq T_2$.

A test case is said to cover an interaction iff the value of every parameter involved in the interaction matches between the test case and interaction. Formally, $\boldsymbol{\sigma} = \langle \sigma_1, \ldots, \sigma_i, \ldots, \sigma_k \rangle$ covers an interaction $T = \{(F_{i_1}, \sigma'_1), \ldots, (F_{i_t}, \sigma'_t)\}$ iff $\sigma_{i_j} = \sigma'_j$ for all $j \in \{i_1, \ldots, i_t\}$. Given a test suite A and interaction T, the set of rows (i.e., test cases) that cover T is

$$\{(F_2, 1), (F_3, 0)\} \quad \{(F_2, 1), (F_4, 3)\}$$
$$\{(F_3, 2), (F_4, 3)\} \quad \{(F_3, 1), (F_4, 3)\}$$

Figure 2.1: Invalid 2-way interactions in the running example

$$\{(F_1, 0), (F_2, 0)\} \quad \{(F_1, 1), (F_3, 2)\} \quad \{(F_1, 0), (F_4, 3)\}$$
$$\{(F_2, 0), (F_3, 0)\} \quad \{(F_2, 1), (F_3, 2)\} \quad \{(F_2, 0), (F_4, 3)\}$$
$$\ldots \text{ (49 in total)}$$

Figure 2.2: Valid 2-way interactions in the running example

signified as $\rho_A(T)$. Moreover, $\rho_A(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \rho_A(T)$ represents the covering test case set of an interaction set $\mathcal{T}$. In other words, $\rho_A(\mathcal{T})$ is the set of rows that cover at least one interaction in $\mathcal{T}$. Note that $\rho_A(\emptyset) = \emptyset$ and $\rho_A(\lambda) = A$.

A test case $\boldsymbol{\sigma}$ is *valid* iff it satisfies the constraint ϕ, i.e., $\phi(\boldsymbol{\sigma}) = true$; otherwise, it is *invalid*. The set of all valid test cases is denoted as $\mathcal{R}$ ($\subseteq S_1 \times S_2 \times \cdots \times S_k$). Hence, $\mathcal{R}$ is referred to as the *exhaustive test suite* consisting of all valid test cases.

The valid/invalid distinction also applies to interactions. Interactions covered by at least one valid test case are *valid*; the other interactions, i.e., those that no valid test cases can cover, are *invalid*. Symbols $\mathcal{I}_t$ and $\mathcal{VI}_t$ denote the sets of all t-way interactions and all valid t-way interactions, respectively. Similarly, this dissertation uses $\overline{\mathcal{I}_t}$ and $\overline{\mathcal{VI}_t}$ to denote the sets of all interactions and of all valid interactions of strength at most t, respectively.

There are 53 2-way interactions in the running example, in which 4 interactions are invalid and 49 are valid. All invalid 2-way interactions in the running example are listed in Figure 2.1. Some valid 2-way interactions are shown in Figure 2.2.

A valid interaction is either *faulty* or *non-faulty*. The outcome of execution of a valid test case is either PASS or FAIL. The outcome is FAIL iff the test case covers one or more

faulty interactions; the outcome is PASS otherwise. The outcome of a test case execution is deterministic. This assumes that if a test case in the model represents a high-level test plan, the actual test cases, which are not modeled, are fixed and do not alter the outcome under the test plan. The outcome of a test suite is the collection of outcomes of all rows in it. In the running example, there are $3 \times 2 \times 3 \times 4 = 72$ test cases in total, of which 48 are valid. In other words, $|\mathcal{R}| = 48$ for the SUT.

2.2 Test Suites in CIT

The mathematical structures of *covering arrays* (CAs), *locating arrays* (LAs), and *detecting arrays* (DAs), are prevalently used as test suites in CIT. The three types of arrays are designed to test an SUT without constraints ($\phi = \textit{true}$ or $\phi = \emptyset$), and they have different testing purposes. CAs contain less test cases than ordinary test suites (i.e., exhaustive test suites, LAs, DAs, etc.), so they can detect the existence of faults with very little cost. LAs and DAs, however, can not only detect faults but also identify faulty interactions by means of test outcomes.

2.2.1 Covering Arrays, Locating Arrays, and Detecting Arrays

A CA has a parameter t that indicates the strength of interactions to be tested. A t-way CA (t-CA) can be formally defined as follows:

$$t\text{-CA} \qquad \forall T \in \mathcal{I}_t : \rho_A(T) \neq \emptyset$$

This condition requires that all t-way interactions T in $\mathcal{I}_t$ will be tested by at least one row in the array. In other words, all t-way faulty interactions, if they exist, can be revealed by a t-CA. The definition of a t-CA also implies that all interactions of strength no larger than t, i.e., $\bar{t}$-way interactions, are covered by at least one test case. That is, a t-CA is also a $(t-1)$-CA when $t > 0$. Thus, a t-CA can also be defined as follows:

t-CA $\quad \forall T \in \overline{\mathcal{I}_t} : \rho_A(T) \neq \emptyset$

The test cost of t-CAs is much less than that of exhaustive test suites because t-CAs do not need to cover all interactions that have strengths larger than t. However, CAs cannot locate or identify faulty interactions correctly. Because t-CAs require that all t-way interactions appear in some test cases, two or more t-way interactions may only appear in the same set of test cases. That is, if some non-faulty interactions only appear in the set of failed test cases, it is impossible to distinguish the non-faulty interactions from the faulty ones using the test outcome.

Meanwhile, LAs and DAs can be used to not only detect the existence of faulty interactions but also identify them. LAs and DAs were first proposed by Colbourn and McClary in [10]. They introduced a total of six types for both LAs and DAs according to fault identification capability. Two types of them exist only in extreme cases. The remaining four types, namely, (d, t)-, $(\overline{d}, t)$-, $(d, \overline{t})$-, $(\overline{d}, \overline{t})$-LAs (and DAs), are as follows ($d \geq 0$, $0 \leq t \leq k$):

(d, t)-LA $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{I}_t$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| = d$:

$\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2) \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$

$(\overline{d}, t)$-LA $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{I}_t$ such that $0 \leq |\mathcal{T}_1| \leq d, 0 \leq |\mathcal{T}_2| \leq d$:

$\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2) \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$

$(d, \overline{t})$-LA $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{I}_t}$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| = d$ and $\mathcal{T}_1, \mathcal{T}_2$ are independent :

$\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2) \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$

$(\overline{d}, \overline{t})$-LA $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{I}_t}$ such that $0 \leq |\mathcal{T}_1| \leq d, 0 \leq |\mathcal{T}_2| \leq d$ and $\mathcal{T}_1, \mathcal{T}_2$ are independent :

$\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2) \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$

(d, t)-DA $\quad \forall \mathcal{T} \subseteq \mathcal{I}_t$ such that $|\mathcal{T}| = d, \forall T \in \mathcal{I}_t$:

$\rho_A(T) \subseteq \rho_A(\mathcal{T}) \Leftrightarrow T \in \mathcal{T}$

$(\overline{d}, t)$-DA $\quad \forall \mathcal{T} \subseteq \mathcal{I}_t$ such that $0 \leq |\mathcal{T}| \leq d, \forall T \in \mathcal{I}_t$:

$\rho_A(T) \subseteq \rho_A(\mathcal{T}) \Leftrightarrow T \in \mathcal{T}$

$(d, \overline{t})$-DA $\quad \forall \mathcal{T} \subseteq \overline{\mathcal{I}_t}$ such that $|\mathcal{T}| = d, \forall T \in \overline{\mathcal{I}_t}$ and $\mathcal{T} \cup \{T\}$ is independent :

$\rho_A(T) \subseteq \rho_A(\mathcal{T}) \Leftrightarrow T \in \mathcal{T}$

$(\overline{d}, \overline{t})$-DA $\quad \forall \mathcal{T} \subseteq \overline{\mathcal{I}_t}$ such that $0 \leq |\mathcal{T}| \leq d, \forall T \in \overline{\mathcal{I}_t}$ and $\mathcal{T} \cup \{T\}$ is independent :

$\rho_A(T) \subseteq \rho_A(\mathcal{T}) \Leftrightarrow T \in \mathcal{T}$

The parameter d of these arrays represents the number of faulty interactions that the arrays can correctly identify, while t represents the strength of the target interactions. Writing $\overline{d}$ or $\overline{t}$ in place of d or t means that the arrays permit at most of d faulty interactions or strength at most t. For instance, a $(1, 2)$-LA or DA can locate one 2-way faulty interaction, while a $(\overline{2}, \overline{5})$-LA or DA can locate at most two faulty interactions with strength no greater than five. When dealing with $\overline{\mathcal{I}_t}$, it is required that $\mathcal{T}_1, \mathcal{T}_2$, or $\mathcal{T} \cup \{T\}$ be independent because if there are $T_1, T_2 \in \overline{\mathcal{I}_t}$ such that $T_1 \subset T_2$, whether T_2 is faulty or not cannot be determined when T_1 is faulty.

The properties of LAs and DAs and the relations among different types of arrays are elaborated on in [10].

2.2.2 Examples of CAs, LAs, and DAs

Examples of CAs, LAs, and DAs are shown in Tables 2.3 and 2.4. Note that there may be different t-CAs of the same sizes (even if all of them are optimal for the given SUT). This also applies to LAs and DAs.

(a) 2-CA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	2	2
$\boldsymbol{\sigma}_3$	0	1	0	3
$\boldsymbol{\sigma}_4$	0	1	1	1
$\boldsymbol{\sigma}_5$	1	0	0	1
$\boldsymbol{\sigma}_6$	1	0	2	3
$\boldsymbol{\sigma}_7$	1	1	0	2
$\boldsymbol{\sigma}_8$	1	1	1	0
$\boldsymbol{\sigma}_9$	2	0	0	3
$\boldsymbol{\sigma}_{10}$	2	0	1	2
$\boldsymbol{\sigma}_{11}$	2	0	2	0
$\boldsymbol{\sigma}_{12}$	2	1	1	3
$\boldsymbol{\sigma}_{13}$	2	1	2	1

(b) 3-CA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	2
$\boldsymbol{\sigma}_3$	0	0	0	3
$\boldsymbol{\sigma}_4$	0	0	1	1
$\boldsymbol{\sigma}_5$	0	0	2	0
$\boldsymbol{\sigma}_6$	0	0	2	2
$\boldsymbol{\sigma}_7$	0	0	2	3
$\boldsymbol{\sigma}_8$	0	1	0	1
$\boldsymbol{\sigma}_9$	0	1	1	0
$\boldsymbol{\sigma}_{10}$	0	1	1	2
$\boldsymbol{\sigma}_{11}$	0	1	1	3
$\boldsymbol{\sigma}_{12}$	0	1	2	1
$\boldsymbol{\sigma}_{13}$	1	0	0	1
$\boldsymbol{\sigma}_{14}$	1	0	1	0
$\boldsymbol{\sigma}_{15}$	1	0	1	2
$\boldsymbol{\sigma}_{16}$	1	0	1	3
$\boldsymbol{\sigma}_{17}$	1	0	2	1
$\boldsymbol{\sigma}_{18}$	1	1	0	0
$\boldsymbol{\sigma}_{19}$	1	1	0	2
$\boldsymbol{\sigma}_{20}$	1	1	0	3
$\boldsymbol{\sigma}_{21}$	1	1	1	1
$\boldsymbol{\sigma}_{22}$	1	1	2	0
$\boldsymbol{\sigma}_{23}$	1	1	2	2
$\boldsymbol{\sigma}_{24}$	1	1	2	3
$\boldsymbol{\sigma}_{25}$	2	0	0	0
$\boldsymbol{\sigma}_{26}$	2	0	0	2
$\boldsymbol{\sigma}_{27}$	2	0	0	3
$\boldsymbol{\sigma}_{28}$	2	0	1	1
$\boldsymbol{\sigma}_{29}$	2	0	2	0
$\boldsymbol{\sigma}_{30}$	2	0	2	3
$\boldsymbol{\sigma}_{31}$	2	1	0	1
$\boldsymbol{\sigma}_{32}$	2	1	1	0
$\boldsymbol{\sigma}_{33}$	2	1	1	2
$\boldsymbol{\sigma}_{34}$	2	1	1	3
$\boldsymbol{\sigma}_{35}$	2	1	2	1
$\boldsymbol{\sigma}_{36}$	2	1	2	2

Figure 2.3: CAs for the running example (constraints ignored)

(a) (1,2)-LA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	1	1
$\boldsymbol{\sigma}_3$	0	0	1	2
$\boldsymbol{\sigma}_4$	0	1	0	3
$\boldsymbol{\sigma}_5$	0	1	2	0
$\boldsymbol{\sigma}_6$	0	1	2	2
$\boldsymbol{\sigma}_7$	1	0	0	1
$\boldsymbol{\sigma}_8$	1	0	1	0
$\boldsymbol{\sigma}_9$	1	0	2	1
$\boldsymbol{\sigma}_{10}$	1	1	0	0
$\boldsymbol{\sigma}_{11}$	1	1	0	2
$\boldsymbol{\sigma}_{12}$	1	1	1	1
$\boldsymbol{\sigma}_{13}$	1	1	2	3
$\boldsymbol{\sigma}_{14}$	2	0	0	2
$\boldsymbol{\sigma}_{15}$	2	0	0	3
$\boldsymbol{\sigma}_{16}$	2	0	1	1
$\boldsymbol{\sigma}_{17}$	2	0	2	3
$\boldsymbol{\sigma}_{18}$	2	1	0	0
$\boldsymbol{\sigma}_{19}$	2	1	1	3

(b) (1,2)-DA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	2
$\boldsymbol{\sigma}_3$	0	0	1	1
$\boldsymbol{\sigma}_4$	0	0	2	3
$\boldsymbol{\sigma}_5$	0	1	0	1
$\boldsymbol{\sigma}_6$	0	1	1	3
$\boldsymbol{\sigma}_7$	0	1	2	0
$\boldsymbol{\sigma}_8$	0	1	2	2
$\boldsymbol{\sigma}_9$	1	0	0	1
$\boldsymbol{\sigma}_{10}$	1	0	1	3
$\boldsymbol{\sigma}_{11}$	1	0	2	0
$\boldsymbol{\sigma}_{12}$	1	0	2	2
$\boldsymbol{\sigma}_{13}$	1	1	0	3
$\boldsymbol{\sigma}_{14}$	1	1	1	0
$\boldsymbol{\sigma}_{15}$	1	1	1	2
$\boldsymbol{\sigma}_{16}$	1	1	2	1
$\boldsymbol{\sigma}_{17}$	2	0	0	3
$\boldsymbol{\sigma}_{18}$	2	0	1	0
$\boldsymbol{\sigma}_{19}$	2	0	1	2
$\boldsymbol{\sigma}_{20}$	2	0	2	1
$\boldsymbol{\sigma}_{21}$	2	1	0	0
$\boldsymbol{\sigma}_{22}$	2	1	0	2
$\boldsymbol{\sigma}_{23}$	2	1	1	1
$\boldsymbol{\sigma}_{24}$	2	1	2	3

Figure 2.4: An LA and DA for the running example (constraints ignored)

2.2.3 Constrained Covering Arrays

The arrays introduced above are designed to test SUTs without constraints. However, real-world systems usually have complicated constraints among parameters. Hence, the constraint handling technique must be considered.

Constrained CAs (CCAs) are the constrained version of CAs. CCAs are the most common form of test suites used in CIT. Most test generation tools for CIT are in effect generators of CCAs.

Definition 1 (CCA)**.** *An array A that consists of valid test cases is a t-CCA iff the following condition holds:*

$$t\text{-CCA} \qquad \forall T \in \mathcal{VI}_t : \rho_A(T) \neq \emptyset$$

The definition of CCAs requires that all *valid* t-way interactions be covered by at least one test case in the test suite. This condition implies that every valid interaction of strength smaller than t is also covered by at least one test case. That is, a t-CCA is also a $(t-1)$-CCA when $t > 0$. Thus, a t-CCA can also be defined as follows:

$$t\text{-CCA} \qquad \forall T \in \overline{\mathcal{VI}_t} : \rho_A(T) \neq \emptyset$$

Figure 2.5a shows a 2-CCA for the running example. It can be observed that none of the invalid interactions, such as $\{(F_2, 1), (F_3, 0)\}$ and $\{(F_2, 1), (F_4, 3)\}$, appear in any rows in Figure 2.5a.

(a) 2-CCA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	3
$\boldsymbol{\sigma}_3$	0	1	1	1
$\boldsymbol{\sigma}_4$	0	1	2	2
$\boldsymbol{\sigma}_5$	1	0	0	2
$\boldsymbol{\sigma}_6$	1	0	0	3
$\boldsymbol{\sigma}_7$	1	0	2	1
$\boldsymbol{\sigma}_8$	1	1	1	0
$\boldsymbol{\sigma}_9$	2	0	0	1
$\boldsymbol{\sigma}_{10}$	2	0	0	3
$\boldsymbol{\sigma}_{11}$	2	0	1	2
$\boldsymbol{\sigma}_{12}$	2	1	2	0

(b) 3-CCA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	1
$\boldsymbol{\sigma}_3$	0	0	0	2
$\boldsymbol{\sigma}_4$	0	0	0	3
$\boldsymbol{\sigma}_5$	0	0	1	1
$\boldsymbol{\sigma}_6$	0	0	2	0
$\boldsymbol{\sigma}_7$	0	1	1	0
$\boldsymbol{\sigma}_8$	0	1	1	2
$\boldsymbol{\sigma}_9$	0	1	2	0
$\boldsymbol{\sigma}_{10}$	0	1	2	1
$\boldsymbol{\sigma}_{11}$	0	1	2	2
$\boldsymbol{\sigma}_{12}$	1	0	0	0
$\boldsymbol{\sigma}_{13}$	1	0	0	1
$\boldsymbol{\sigma}_{14}$	1	0	0	2
$\boldsymbol{\sigma}_{15}$	1	0	0	3
$\boldsymbol{\sigma}_{16}$	1	0	1	0
$\boldsymbol{\sigma}_{17}$	1	0	2	2
$\boldsymbol{\sigma}_{18}$	1	1	1	1
$\boldsymbol{\sigma}_{19}$	1	1	1	2
$\boldsymbol{\sigma}_{20}$	1	1	2	0
$\boldsymbol{\sigma}_{21}$	1	1	2	1
$\boldsymbol{\sigma}_{22}$	2	0	0	0
$\boldsymbol{\sigma}_{23}$	2	0	0	1
$\boldsymbol{\sigma}_{24}$	2	0	0	2
$\boldsymbol{\sigma}_{25}$	2	0	0	3
$\boldsymbol{\sigma}_{26}$	2	0	1	0
$\boldsymbol{\sigma}_{27}$	2	0	1	2
$\boldsymbol{\sigma}_{28}$	2	0	2	1
$\boldsymbol{\sigma}_{29}$	2	1	1	0
$\boldsymbol{\sigma}_{30}$	2	1	1	1
$\boldsymbol{\sigma}_{31}$	2	1	2	0
$\boldsymbol{\sigma}_{32}$	2	1	2	2

Figure 2.5: A 2-CCA and 3-CCA for the running example

CHAPTER 3

CONSTRAINED LOCATING ARRAYS

Incorporating constraints into LAs is not as straightforward as for CAs. In addition to the existence of invalid interactions, constraints may cause some valid interactions to be no longer identifiable from others. This fact consequently makes the definition of LAs no longer valid for SUTs with constraints. To solve such a problem, the non-identifiable interactions should be clarified first. Then, the definition of LAs must be relaxed so that the constrained versions of LAs can be applied to SUTs with constraints.

3.1 Definitions

LAs identify faulty interactions with test outcomes. In an LA, a set of interactions is uniquely mapped to a set of test cases. Hence, when a test suite is executed, a set of faulty interactions can be uniquely identified according to the failed test cases. However, with the existence of constraints, a set of interactions may be restricted to appear in the same set of test cases, which is mapped to another set of interactions. Consequently, LAs cannot be applied to the SUT. In the running example, the case can be illustrated as follows.

Constraint ϕ_2 in Table 2.1 enforces every test case that has the parameter-value pair $(F_4, 3)$ to always contain the parameter-value pairs of $(F_2, 0)$ and $(F_3, 0)$. Then, the set of

the single interaction $\mathcal{T}_a = \{\{(F_2, 0), (F_4, 3)\}\}$ will always have the same covering test cases as another set of the single interaction $\mathcal{T}_b = \{\{(F_3, 0), (F_4, 3)\}\}$. The definition of $(1, 2)$ (or $(\overline{1}, 2)$, $(1, \overline{2})$, and $(\overline{1}, \overline{2})$)-LAs cannot be held for the running example.

The relation between two such sets of interactions is called *indistinguishability*. The formal definition is as follows.

Definition 2. *A pair of sets of valid interactions, $\mathcal{T}_a$ and $\mathcal{T}_b$, are distinguishable iff*

$$\rho_{\mathcal{R}}(\mathcal{T}_a) \neq \rho_{\mathcal{R}}(\mathcal{T}_b).$$

If $\mathcal{T}_a$ and $\mathcal{T}_b$ are distinguishable, it is denoted as $\mathcal{T}_a \not\sim \mathcal{T}_b$; otherwise, it is denoted as $\mathcal{T}_a \sim \mathcal{T}_b$.

Note that even if no constraints exist, there can be some indistinguishable pairs of interaction sets. In the running example, the two interaction sets $\mathcal{T}_c = \{\{(F_1, 0)\}, \{(F_1, 1)\}, \{(F_1, 2)\}\}$ and $\mathcal{T}_d = \{\{(F_3, 0)\}, \{(F_3, 1)\}, \{(F_3, 2)\}\}$ are indistinguishable even if the constraints are omitted because the covering test sets for both interaction sets are $\mathcal{R}$. If two sets of interactions are indistinguishable, there is no other method to distinguish them from each other according to test outcomes.

Based on the definition of distinguishable interaction sets, (d, t)-, $(\overline{d}, t)$-, $(d, \overline{t})$-, and $(\overline{d}, \overline{t})$-CLAs, the constrained versions of the corresponding LAs, can be defined as follows.

Definition 3. *Let $d \geq 0$ and $0 \leq t \leq k$. Let $\mathcal{VI}_t$ be the set of all valid t-way interactions and $\overline{\mathcal{VI}_t}$ be the set of all valid interactions of strength at most t. An array A that consists of valid test cases or no rows is a (d, t)-, $(\overline{d}, t)$-, $(d, \overline{t})$- or $(\overline{d}, \overline{t})$-CLA iff the corresponding*

condition below holds:

(d,t)-*CLA* $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_t$ *such that* $|\mathcal{T}_1| = |\mathcal{T}_2| = d$:

$\mathcal{T}_1 \not\sim \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

$(\overline{d},t)$-*CLA* $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_t$ *such that* $0 \leq |\mathcal{T}_1|, |\mathcal{T}_2| \leq d$:

$\mathcal{T}_1 \not\sim \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

$(d,\overline{t})$-*CLA* $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ *such that* $|\mathcal{T}_1| = |\mathcal{T}_2| = d$ *and* $\mathcal{T}_1, \mathcal{T}_2$ *are independent* :

$\mathcal{T}_1 \not\sim \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

$(\overline{d},\overline{t})$-*CLA* $\quad \forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ *such that* $0 \leq |\mathcal{T}_1|, |\mathcal{T}_2| \leq d$ *and* $\mathcal{T}_1, \mathcal{T}_2$ *are independent* :

$\mathcal{T}_1 \not\sim \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

(in extreme cases where no two such interaction sets $\mathcal{T}_1, \mathcal{T}_2$ *exist, any* A *is a CLA)*

The intuition of the definition is that if the SUT has a set of d (or $\leq d$) faulty interactions, then the test outcome obtained by executing all test cases in A will be different from the one that would be obtained if the SUT had a different set of d (or $\leq d$) faulty interactions, unless the two interaction sets are not distinguishable.

3.2 Examples

Examples of a $(\overline{1},\overline{1})$- and $(\overline{2},\overline{1})$-CLA for the running example are shown in Table 3.1. Table 3.2 shows a $(\overline{1},\overline{2})$-CLA for the running example. Given an SUT, the minimum CLA and minimum LA for the SUT (constraints ignored) may be different in size even when d and t are the same; constraints may cause the sizes of CLAs to increase or decrease.

3.3 Properties of CLAs

Observation 1. *A* $(\overline{d},\overline{t})$*-CLA is a* $(\overline{d},t)$*- and* $(d,\overline{t})$*-CLA. A* $(\overline{d},t)$*-CLA and* $(d,\overline{t})$*-CLA are both a* (d,t)*-CLA. A* $(\overline{d},\overline{t})$*-CLA and* $(\overline{d},t)$*-CLA are a* $(\overline{d-1},\overline{t})$*-CLA and* $(\overline{d-1},t)$*-*

(a) $(\overline{1}, \overline{1})$-CLA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	3
$\boldsymbol{\sigma}_2$	0	1	2	1
$\boldsymbol{\sigma}_3$	1	0	0	2
$\boldsymbol{\sigma}_4$	1	0	1	1
$\boldsymbol{\sigma}_5$	2	0	0	1
$\boldsymbol{\sigma}_6$	2	1	1	0

(b) $(\overline{2}, \overline{1})$-CLA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	1	2
$\boldsymbol{\sigma}_3$	0	0	2	1
$\boldsymbol{\sigma}_4$	1	1	1	0
$\boldsymbol{\sigma}_5$	2	0	0	3
$\boldsymbol{\sigma}_6$	2	1	2	2

Figure 3.1: A $(\overline{1}, \overline{1})$-CLA and $(\overline{2}, \overline{1})$-CLA for the running example

CLA, respectively. A $(\overline{d}, \overline{t})$-CLA and $(d, \overline{t})$-CLA are a $(\overline{d}, \overline{t-1})$-CLA and $(d, \overline{t-1})$-CLA, respectively.

Observation 2. *Suppose that the SUT has no constraints, i.e., $\phi(\boldsymbol{\sigma}) = true$ for all $\boldsymbol{\sigma} \in V_1 \times \ldots \times V_k$, and that an LA A exists. Then, 1) A is a CLA with the same parameters, and 2) any CLA with the same parameters as A is an LA (which is possibly different from A) with the same parameters.*

Lemma 1. *A pair of sets of valid interactions, $\mathcal{T}_1$ and $\mathcal{T}_2$, are distinguishable iff there is a valid test case that covers some interaction in $\mathcal{T}_1$ or $\mathcal{T}_2$ but no interactions in $\mathcal{T}_2$ or $\mathcal{T}_1$, respectively, i.e., for some valid test case $\boldsymbol{\sigma} \in \mathcal{R}$, $(\exists T \in \mathcal{T}_1 : T \subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma})$ or $(\exists T \in \mathcal{T}_2 : T \subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma})$.*

Proof. (If part) Suppose that there is such a valid test case $\boldsymbol{\sigma}$. Consider an array A that contains $\boldsymbol{\sigma}$. Then, either $\boldsymbol{\sigma} \in \rho_A(\mathcal{T}_1) \wedge \boldsymbol{\sigma} \notin \rho_A(\mathcal{T}_2)$ or $\boldsymbol{\sigma} \notin \rho_A(\mathcal{T}_1) \wedge \boldsymbol{\sigma} \in \rho_A(\mathcal{T}_2)$; thus, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. (Only if part) Suppose that there is no such valid test case, i.e., for every valid test case $\boldsymbol{\sigma}$, $(\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma}) \vee (\exists T \in \mathcal{T}_2 : T \subseteq \boldsymbol{\sigma})$ and $(\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma}) \vee (\exists T \in \mathcal{T}_1 : T \subseteq \boldsymbol{\sigma})$. This means that for every valid test case $\boldsymbol{\sigma}$,

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	3
$\boldsymbol{\sigma}_3$	0	0	1	1
$\boldsymbol{\sigma}_4$	0	1	1	2
$\boldsymbol{\sigma}_5$	0	1	2	0
$\boldsymbol{\sigma}_6$	1	0	0	2
$\boldsymbol{\sigma}_7$	1	0	0	3
$\boldsymbol{\sigma}_8$	1	0	1	2
$\boldsymbol{\sigma}_9$	1	1	1	1
$\boldsymbol{\sigma}_{10}$	1	1	2	0
$\boldsymbol{\sigma}_{11}$	1	1	2	2
$\boldsymbol{\sigma}_{12}$	2	0	0	1
$\boldsymbol{\sigma}_{13}$	2	0	0	3
$\boldsymbol{\sigma}_{14}$	2	0	2	0
$\boldsymbol{\sigma}_{15}$	2	1	1	0
$\boldsymbol{\sigma}_{16}$	2	1	1	2
$\boldsymbol{\sigma}_{17}$	2	1	2	1

Figure 3.2: A $(\overline{1}, \overline{2})$-CLA for the running example

$(\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma})$ or $(\exists T \in \mathcal{T}_1 : T \subseteq \boldsymbol{\sigma}) \vee (\exists T \in \mathcal{T}_2 : T \subseteq \boldsymbol{\sigma})$. Hence, for any test case $\boldsymbol{\sigma}$ in A, $\boldsymbol{\sigma} \notin \rho_A(\mathcal{T}_1) \wedge \boldsymbol{\sigma} \notin \rho_A(\mathcal{T}_2)$ or $\boldsymbol{\sigma} \in \rho_A(\mathcal{T}_1) \wedge \boldsymbol{\sigma} \in \rho_A(\mathcal{T}_2)$. As a result, for any A, $\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2)$. □

Theorem 2. *If A is an array consisting of all valid test cases, then A is a CLA with any parameters.*

Proof. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be any interaction sets that are distinguishable. By Lemma 1, a valid test case $\boldsymbol{\sigma}$ exists such that $(\exists T \in \mathcal{T}_1 : T \subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma})$ or $(\exists T \in \mathcal{T}_2 : T \subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma})$. Because A contains this test case and by the same argument of the proof of the if-part of Lemma 1, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. □

Theorem 3. *Let t be an integer such that $0 \leq t < k$. If an $N \times k$ array A is a $(t+1)$-CCA, then A is also a $(\overline{1}, \overline{t})$-CLA.*

Proof. Recall that an array A is a $(\overline{1}, \overline{t})$-CLA iff $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ for all $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ such that $0 \leq |\mathcal{T}_1| \leq 1$, $0 \leq |\mathcal{T}_2| \leq 1$, and $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinguishable (see Definition 2; note that $\mathcal{T}_1$ and $\mathcal{T}_2$ are independent because they contain at most one interaction).

Now suppose that an $N \times k$ array A is a $(t+1)$-CCA such that $0 \leq t < k$. If $|\mathcal{T}_1| = |\mathcal{T}_2| = 0$, then $\mathcal{T}_1 = \mathcal{T}_2 = \emptyset$, and thus they are not distinguishable. If $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 0$, then $\rho_A(\mathcal{T}_1) \neq \emptyset$ because A is a $(t+1)$-CCA, and thus any $T \in \overline{\mathcal{VI}_{t+1}}$ is covered by some row in A. Because $\rho_A(\emptyset) = \emptyset$, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2) = \emptyset$ holds for any $\mathcal{T}_1$, $\mathcal{T}_2 \subseteq \overline{\mathcal{VT}_t}$ if $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 0$. The same argument clearly holds if $|\mathcal{T}_1| = 0$ and $|\mathcal{T}_2| = 1$.

For the remainder of the proof, the case in which $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 1$ is considered. This proof will show that $\rho_A(T_a) \neq \rho_A(T_b)$ (i.e., $\rho_A(\{T_a\}) \neq \rho_A(\{T_b\})$) always holds for any $T_a, T_b \in \overline{\mathcal{VI}_t}$ if $\{T_a\}$ and $\{T_b\}$ are distinguishable. Let $T_a = \{(F_{a_1}, u_{a_1}), \ldots, (F_{a_l}, u_{a_l})\}$ and $T_b = \{(F_{b_1}, v_{b_1}), \ldots, (F_{b_m}, v_{b_m})\}$ $(0 \leq l, m \leq t)$. Also,

let $\mathrm{F} = \{F_{a_1}, \ldots, F_{a_l}\} \cap \{F_{b_1}, \ldots, F_{b_m}\}$, *i.e.*, F is the set of factors that are involved in both interactions. There are two cases to consider:

(1) For some $F_i \in \mathrm{F}, u_i \neq v_i$. That is, the two interactions have different values on some factor F_i. In this case, T_a and T_b never occur in the same test case. Because A is a $(t+1)$-CCA, $\rho_A(T_a) \neq \emptyset$ and $\rho_A(T_b) \neq \emptyset$. Hence, $\rho_A(T_a) \neq \rho_A(T_b)$.

(2) $\mathrm{F} = \emptyset$ or for all $F_i \in \mathrm{F}, u_i = v_i$. That is, the two interactions have no common factors or have the same value for every factor in common. Because $\{T_a\}$ and $\{T_b\}$ are distinguishable, there must be at least one valid test case $\boldsymbol{\sigma}$ in $\mathcal{R}$ that covers either T_a or T_b but not both. Suppose that $\boldsymbol{\sigma}$ covers T_a but does not cover T_b. In this case, there is a factor $F_j \in \{F_{b_1}, \ldots, F_{b_m}\} \backslash \mathrm{F}$ such that the value on F_j of $\boldsymbol{\sigma}$, denoted by w_j, is different from v_j because otherwise, T_b would be covered by $\boldsymbol{\sigma}$. Now, consider an $(l+1)$-way interaction $T'_a = T_a \cup \{(F_j, w_j)\}$. Because the valid test case $\boldsymbol{\sigma}$ covers T'_a, T'_a is a $(l+1)$-way valid interaction. Because A is a $(t+1)$-CCA and $l+1 \leq t+1$, A contains at least one row that covers T'_a. This row covers T_a but does not cover T_b because the value on F_j is w_j and $w_j \neq v_j$. Hence, $\rho_A(T_a) \neq \rho_A(T_b)$. The same argument applies to the case in which $\boldsymbol{\sigma}$ covers T_b but not T_a

As a result, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ holds for any $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VT}_t}$ if $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$, and they are distinguishable. □

Lemma 4. *Suppose that an $N \times k$ array A is a $(\overline{1}, t)$-CLA such that $1 \leq t \leq k$. Then, A is a t-CCA.*

Proof. Because A is a $(\overline{1}, t)$-CLA, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ for any $\mathcal{T}_1, \mathcal{T}_2 (\neq \mathcal{T}_1) \subseteq \mathcal{VI}_t$ such that $|\mathcal{T}_1|, |\mathcal{T}_2| \leq 1$. Hence, if $\mathcal{T}_1 = \emptyset$ and $\mathcal{T}_2 = \{T\}$ for any $T \in \mathcal{VI}_t$, then $\rho_A(\mathcal{T}_1) = \rho_A(\emptyset) = \emptyset \neq \rho_A(\mathcal{T}_2) = \rho_A(T)$. □

Theorem 5. *If an $N \times k$ array A is a $(\overline{1}, t)$-CLA such that $1 \leq t \leq k$, then A is a $(\overline{1}, \bar{t})$-CLA.*

Proof. Suppose that A is a $(\overline{1}, t)$-CLA such that $1 \leq t \leq k$. By Lemma 4, A is a t-CCA; thus, by Theorem 3, it is a $(\overline{1}, \overline{t-1})$-CLA. Recall that A is a $(\overline{1}, \bar{t})$-CLA iff $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ for all $\mathcal{T}_1, \mathcal{T}_2 \in \overline{\mathcal{VI}_t}$ such that $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinguishable and $0 \leq |\mathcal{T}_1|, |\mathcal{T}_2| \leq 1$ (note that $\mathcal{T}_1$ and $\mathcal{T}_2$ are trivially independent). If $|\mathcal{T}_1| = |\mathcal{T}_2| = 0$, then $\mathcal{T}_1$ and $\mathcal{T}_2$ are both $\emptyset$ and thus indistinguishable. If $|\mathcal{T}_1| = 0$ and $|\mathcal{T}_2| = 1$, then $\mathcal{T}_2 = \{T\}$ for some $T \in \overline{\mathcal{VI}_t}$. Because A is a t-CCA, $\rho_A(\{T\}) \neq \emptyset$ for any $T \in \overline{\mathcal{VI}_t}$. Therefore, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. Clearly, this argument holds when $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 0$.

In the following part, the proof assumes that $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$. Let $\mathcal{T}_1 = \{T_a\}$ and $\mathcal{T}_2 = \{T_b\}$, where $T_a, T_b \in \overline{\mathcal{VI}_t}$. Without loss of generality, the proof assumes that the strength of T_a is at most equal to that of T_b, i.e., $0 \leq |T_a| \leq |T_b| \leq t$. If $0 \leq |T_a| \leq |T_b| \leq t-1$ and $\{T_a\}$ and $\{T_b\}$ are distinguishable, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ because A is a $(\overline{1}, \overline{t-1})$-CLA. If $|T_a| = |T_b| = t$ and $\{T_a\}$ and $\{T_b\}$ are distinguishable, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ because A is a $(\overline{1}, t)$-CLA.

Now consider the remaining case where $0 \leq |T_a| < |T_b| = t$. Assume that $\{T_a\}$ and $\{T_b\}$ are distinguishable. Below shows that $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ under this assumption. Because of the assumption, at least one of the following two cases holds. Case 1: for some $\boldsymbol{\sigma} \in \mathcal{R}$, $T_a \subseteq \sigma$ and $T_b \not\subseteq \boldsymbol{\sigma}$. Case 2: for some $\boldsymbol{\sigma} \in \mathcal{R}$, $T_a \not\subseteq \boldsymbol{\sigma}$ and $T_b \subseteq \boldsymbol{\sigma}$.

Let $T_a = \{(F_{a_1}, u_{a_1}), \ldots, (F_{a_l}, u_{a_l})\}$ and $T_b = \{(F_{b_1}, v_{b_1}), \ldots, (F_{b_t}, v_{b_t})\}$ $(0 \leq l \leq t-1)$. Also, let $\mathrm{F} = \{F_{a_1}, \ldots, F_{a_l}\} \cap \{F_{b_1}, \ldots, F_{b_t}\}$, i.e., F is the set of factors that are involved in both interactions.

Case 1: Let $\boldsymbol{\sigma}_1$ be any test case in $\mathcal{R}$ such that $T_a \subseteq \boldsymbol{\sigma}_1$ and $T_b \not\subseteq \boldsymbol{\sigma}_1$. Choose a factor F_{b_i}, $1 \leq i \leq t$, such that the value on F_{b_i} in $\boldsymbol{\sigma}_1$ is different from v_{b_i}. Such a factor must always exist because otherwise, $T_b \subseteq \boldsymbol{\sigma}_1$. Let w_{b_i} denote the value on F_{b_i} in $\boldsymbol{\sigma}_1$. Then, the interaction $\hat{T} = T_a \cup \{(F_{b_i}, w_{b_i})\}$ is covered by $\boldsymbol{\sigma}_1$ ($\hat{T} \subseteq \boldsymbol{\sigma}_1$) and thus is valid. The

strength of $\hat{T}$ is l (if $F_{b_i} \in \mathrm{F}$, in which case $u_{b_i} = w_{b_i}$) or $l+1$ (if $F_{b_i} \notin \mathrm{F}$). For any test case $\boldsymbol{\sigma} \in \mathcal{R}$, $\hat{T} \subseteq \boldsymbol{\sigma} \Rightarrow T_b \not\subseteq \boldsymbol{\sigma}$ holds because $w_{b_i} \neq v_{b_i}$. Because A is a t-CCA and the strength of $\hat{T}$ is at most t, A has a row that covers $\hat{T}$. This row covers T_a but not T_b; thus, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$.

Case 2: Let $\boldsymbol{\sigma}_2$ be any test case in $\mathcal{R}$ such that $T_a \not\subseteq \boldsymbol{\sigma}_2$ and $T_b \subseteq \boldsymbol{\sigma}_2$. Also, let $\check{T}$ be any t-way interaction such that $\check{T} = T_a \cup \{(F_{b_{i_1}}, v_{b_{i_1}}), \ldots, (F_{b_{i_{t-l}}}, v_{b_{i_{t-l}}})\}$ for some $F_{b_{i_1}}, \ldots, F_{b_{i_{t-l}}} \notin F$. In other words, $\check{T}$ is a t-way interaction that is obtained by extending T_a with some $t-l$ factor-value pairs in T_b.

If $\check{T}$ is valid, then $\{\check{T}\}$ and $\{T_b\}$ are distinguishable because $T_b \subseteq \boldsymbol{\sigma}_2$ and $\check{T} \not\subseteq \boldsymbol{\sigma}_2$ (because $T_a \not\subseteq \boldsymbol{\sigma}_2$ and $T_a \subseteq \check{T}$). A is a $(\overline{1}, t)$-CLA; thus, A must have a row $\boldsymbol{r}$ that covers either $\check{T}$ or T_b, i.e., $\check{T} \subseteq r \wedge T_b \not\subseteq \boldsymbol{r}$ or $\check{T} \not\subseteq \boldsymbol{r} \wedge T_b \subseteq \boldsymbol{r}$. $\check{T} \subseteq \boldsymbol{r} \wedge T_b \not\subseteq \boldsymbol{r}$ directly implies $T_a \subseteq \boldsymbol{r} \wedge T_b \not\subseteq \boldsymbol{r}$, while $\check{T} \not\subseteq \boldsymbol{r} \wedge T_b \subseteq \boldsymbol{r}$ implies $\check{T} \backslash T_b \not\subseteq \boldsymbol{r}$, which means $T_a \not\subseteq \boldsymbol{r}$. Hence, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$.

If $\check{T}$ is not valid, then it can be shown that T_a and T_b never appear simultaneously in any test case $\boldsymbol{\sigma} \in \mathcal{R}$ as follows. If there is some test case $\boldsymbol{\sigma}$ in $\mathcal{R}$ in which T_a and T_b are both covered, then $\check{T}$ is also covered by some test cases (including $\boldsymbol{\sigma}$) in $\mathcal{R}$, i.e., $\check{T}$ is valid. The contraposition of this argument is that if $\check{T}$ is invalid, then there is no test case in $\mathcal{R}$ that covers T_a and T_b. Because A is a t-CCA and $T_a, T_b \in \overline{\mathcal{VI}_t}$, $\rho_A(T_a) \neq \emptyset$ and $\rho_A(T_b) \neq \emptyset$. Hence, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. □

CHAPTER 4

CONSTRAINED DETECTING ARRAYS

Similar to LAs, applying the constraint handling technique to DAs is not direct. In addition to the existence of invalid interactions, the conditions required by the definition of DAs cannot always be satisfied because of constraints. In this chapter, this problem is explained. Then, the dissertation proposes CDAs as the constrained version of DAs. Finally, some examples of CDAs and proofs on properties of CDAs follow.

4.1 Definitions

Based on the definition of DAs, faulty interactions are identified with patterns of failed and passed test cases. This holds because of the requirements in the definition of DAs, i.e., $\rho_A(T) \subseteq \rho_A(\mathcal{T}) \Leftrightarrow T \in \mathcal{T}$ (where T and $\mathcal{T}$ are a t- or $\bar{t}$-way interaction and a set of t- or $\bar{t}$-way interactions, respectively and A is an array). In other words, if an interaction T is not included in an interaction set $\mathcal{T}$, then the covering test set of the interaction T must not be a subset of the covering test set of the interaction set $\mathcal{T}$. Thus, different interaction sets are mapped to different sets of test cases in a strict manner. However, such mapping cannot be guaranteed if constraints exist.

This problem can be described using the running example in Table 2.1. Let $\mathcal{T} =$

$\{T_1\} = \{\{(F_2, 0), (F_4, 0)\}\}$ be the set of one 2-way interaction and $T_2 = \{(F_3, 0), (F_4, 0)\}$ be a 2-way interaction. Because of the constraint ϕ_1, every valid test case that covers T_2 also covers T_1, but there exist valid test cases that cover T_1 but not T_2. In other words, $\rho_A(T_2)$ is a subset of $\rho_A(\mathcal{T}) = \{\rho_A(T_1)\}$ as long as the array A contains only valid test cases. Therefore, the definition of DAs no longer holds in the SUT.

Moreover, assume that T_1 is the faulty interaction; then, all test cases in $\rho_A(\mathcal{T})$ will fail. That is, all test cases in $\rho_A(T_2)$ will fail. In this case, a tester can determine that T_1 is the faulty interaction, while they cannot determine whether T_2 is also faulty. The relation between an interaction set and an interaction above is called *masking* in this dissertation.

Definition 4. *A set $\mathcal{T}$ of valid interactions* masks *a valid interaction T iff $T \notin \mathcal{T}$ and*

$$\forall \boldsymbol{\sigma} \in \mathcal{R} : T \subseteq \boldsymbol{\sigma} \Rightarrow (\exists T' \in \mathcal{T} : T' \subseteq \boldsymbol{\sigma}).$$

If $\mathcal{T}$ masks T, it is denoted as $\mathcal{T} \succ T$; otherwise, it is denoted as $\mathcal{T} \not\succ T$. By definition, $\mathcal{T} \not\succ T$ iff $T \in \mathcal{T}$ or

$$\exists \boldsymbol{\sigma} \in \mathcal{R} : T \subseteq \sigma \wedge (\forall T' \in \mathcal{T} : T' \not\subseteq \sigma).$$

Based on the definition of masking, (d, t)-, $(\overline{d}, t)$-, $(d, \overline{t})$-, and $(\overline{d}, \overline{t})$-CDAs, the constrained versions of the corresponding DAs, are defined as follows.

Definition 5. *Let $d \geq 0$ and $0 \leq t \leq k$. An array A that consists of valid test cases or no rows is a (d, t)-, $(\overline{d}, t)$-, $(d, \overline{t})$- or $(\overline{d}, \overline{t})$-CDA iff the corresponding condition shown below*

holds:

(d,t)-*CDA* $\quad \forall \mathcal{T} \subseteq \mathcal{VI}_t$ *such that* $|\mathcal{T}| = d, \forall T \in \mathcal{VI}_t$:

$$\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$$

$(\overline{d},t)$-*CDA* $\quad \forall \mathcal{T} \subseteq \mathcal{VI}_t$ *such that* $0 \leq |\mathcal{T}| \leq d, \forall T \in \mathcal{VI}_t$:

$$\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$$

$(d,\overline{t})$-*CDA* $\quad \forall \mathcal{T} \subseteq \overline{\mathcal{VI}_t}$ *such that* $|\mathcal{T}| = d, \forall T \in \overline{\mathcal{VI}_t}$ *and* $\mathcal{T} \cup \{T\}$ *is independent* :

$$\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$$

$(\overline{d},\overline{t})$-*CDA* $\quad \forall \mathcal{T} \subseteq \overline{\mathcal{VI}_t}$ *such that* $0 \leq |\mathcal{T}| \leq d, \forall T \in \overline{\mathcal{VI}_t}$ *and* $\mathcal{T} \cup \{T\}$ *is independent* :

$$\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$$

4.2 Examples

Examples of a $(\overline{1},\overline{1})$- and $(\overline{2},\overline{1})$-CDA for the running example are shown in Table 4.1. Table 4.2 shows a $(\overline{1},\overline{2})$-CDA for the running example. Given an SUT, the minimum CDA and the minimum DA for the SUT (constraints ignored) with the same d and t may have different sizes. Depending on constraints, the minimum sizes of CDAs may be larger or smaller than those of DAs.

4.3 Properties of CDAs

Observation 3. *A* $(\overline{d},\overline{t})$*-CDA is a* $(\overline{d},t)$*-CDA and* $(d,\overline{t})$*-CDA. A* $(\overline{d},t)$*-CDA and* $(d,\overline{t})$*-CDA are both a* (d,t)*-CDA. When* $d > 0$*, a* $(\overline{d},\overline{t})$*-CDA and* $(\overline{d},t)$*-CDA are a* $(\overline{d-1},\overline{t})$*-CDA and* $(\overline{d-1},t)$*-CDA, respectively. When* $t > 0$*, a* $(\overline{d},\overline{t})$*-CDA and* $(d,\overline{t})$*-CDA are a* $(\overline{d},\overline{t-1})$*-CDA and* $(d,\overline{t-1})$*-CDA, respectively.*

Observation 4. *Suppose that the SUT has no constraints, i.e.,* $\phi(\boldsymbol{\sigma}) = true$ *for all* $\boldsymbol{\sigma} \in \mathcal{R} = V_1 \times V_2 \times \cdots \times V_k$*, and that a* (d,t)*-DA* A *exists. Then,* A *is a* (d,t)*-CDA. This also*

(a) $(\overline{1}, \overline{1})$-CDA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	1	2	1
$\boldsymbol{\sigma}_3$	1	0	0	3
$\boldsymbol{\sigma}_4$	1	0	1	1
$\boldsymbol{\sigma}_5$	1	1	1	2
$\boldsymbol{\sigma}_6$	2	0	0	3
$\boldsymbol{\sigma}_7$	2	0	2	2
$\boldsymbol{\sigma}_8$	2	1	1	0

(b) $(\overline{2}, \overline{1})$-CDA

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	3
$\boldsymbol{\sigma}_3$	0	0	2	1
$\boldsymbol{\sigma}_4$	0	1	1	2
$\boldsymbol{\sigma}_5$	0	1	2	0
$\boldsymbol{\sigma}_6$	1	0	1	1
$\boldsymbol{\sigma}_7$	1	0	0	1
$\boldsymbol{\sigma}_8$	1	0	0	2
$\boldsymbol{\sigma}_9$	1	0	0	3
$\boldsymbol{\sigma}_{10}$	1	1	1	0
$\boldsymbol{\sigma}_{11}$	1	1	2	1
$\boldsymbol{\sigma}_{12}$	2	0	0	3
$\boldsymbol{\sigma}_{13}$	2	0	1	2
$\boldsymbol{\sigma}_{14}$	2	0	2	0
$\boldsymbol{\sigma}_{15}$	2	1	1	1
$\boldsymbol{\sigma}_{16}$	2	1	2	2

Figure 4.1: A $(\overline{1}, \overline{1})$-CDA and $(\overline{2}, \overline{1})$-CDA for the running example

	F_1	F_2	F_3	F_4
$\boldsymbol{\sigma}_1$	0	0	0	0
$\boldsymbol{\sigma}_2$	0	0	0	1
$\boldsymbol{\sigma}_3$	0	0	0	3
$\boldsymbol{\sigma}_4$	0	0	1	1
$\boldsymbol{\sigma}_5$	0	0	2	2
$\boldsymbol{\sigma}_6$	0	1	1	2
$\boldsymbol{\sigma}_7$	0	1	2	0
$\boldsymbol{\sigma}_8$	0	1	2	1
$\boldsymbol{\sigma}_9$	1	0	0	2
$\boldsymbol{\sigma}_{10}$	1	0	0	3
$\boldsymbol{\sigma}_{11}$	1	0	1	0
$\boldsymbol{\sigma}_{12}$	1	0	2	1
$\boldsymbol{\sigma}_{13}$	1	1	1	1
$\boldsymbol{\sigma}_{14}$	1	1	2	0
$\boldsymbol{\sigma}_{15}$	1	1	2	2
$\boldsymbol{\sigma}_{16}$	2	0	0	0
$\boldsymbol{\sigma}_{17}$	2	0	0	1
$\boldsymbol{\sigma}_{18}$	2	0	0	2
$\boldsymbol{\sigma}_{19}$	2	0	0	3
$\boldsymbol{\sigma}_{20}$	2	0	1	2
$\boldsymbol{\sigma}_{21}$	2	0	2	0
$\boldsymbol{\sigma}_{22}$	2	1	1	0
$\boldsymbol{\sigma}_{23}$	2	1	1	1
$\boldsymbol{\sigma}_{24}$	2	1	2	2

Figure 4.2: A $(\overline{1}, \overline{2})$-CDA for the running example

applies when d or t is replaced with $\overline{d}$ or $\overline{t}$, respectively.

Theorem 6. *For $d \leq \tau_t$, a (d,t)-CDA is equivalent to a $(\overline{d},t)$-CDA.*

Proof. Trivially, a $(0,t)$-CDA is a $(\overline{0},t)$-CDA. Let A be a (d,t)-CDA such that $1 \leq d \leq \tau_t$ and $t > 0$. The following will show that A is a $(d-1,t)$-CDA. Let $\mathcal{T}$ and T be a set of $d-1$ valid interactions of strength t and a t-way valid interaction, respectively. If $\mathcal{T} \succ T$ or $T \in \mathcal{T}$, then $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$ holds trivially. The remainder of the proof considers the case where $T \notin \mathcal{T}$ and $\mathcal{T} \not\succ T$. In this case, there is some $\sigma \in \mathcal{R}$ such that $T \subseteq \sigma$ and $\sigma \notin \rho_{\mathcal{R}}(\mathcal{T})$. Because $|\mathcal{T} \cup \{T\}| \leq \tau_t$, $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T} \cup \{T\})$ is not empty. Let T' be any t-way interaction that appears in a test case in $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T} \cup \{T\})$ and has exactly the same t parameters as T. Note that T and T' cannot appear in any test case simultaneously. Let $\mathcal{T}' = \mathcal{T} \cup \{T'\}$. $\mathcal{T}' \not\succ T$ because $T \subseteq \sigma$, $\sigma \notin \rho_{\mathcal{R}}(\mathcal{T})$, and $\sigma \notin \rho_{\mathcal{R}}(T')$. Because A is a (d,t)-CDA and $\mathcal{T}' \not\succ T$, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T}')$. Hence, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$, which means that $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$. By induction, A is a (d',t)-CDA for any $0 \leq d' \leq d$, and thus is a $(\overline{d},t)$-CDA. □

Theorem 7. *For $d = t = 0$ or $d \leq \tau_1$ and $t > 0$, a $(d,\overline{t})$-CDA is equivalent to a $(\overline{d},\overline{t})$-CDA.*

Proof. Trivially, $(0,\overline{t})$-CDA is a $(\overline{0},\overline{t})$-CDA. Let A be a $(d,\overline{t})$-CDA such that $1 \leq d \leq \tau_t$ and $t > 0$. Below, the proof will show that A is a $(d-1,\overline{t})$-CDA. Let $\mathcal{T} \subseteq \overline{\mathcal{VI}_t}$ such that $\mathcal{T}$ is independent and $|\mathcal{T}| = d-1$. Let T be a valid interaction of strength at most t. If $\mathcal{T} \succ T$ or $T \in \mathcal{T}$, then $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$ holds trivially.

Consider the remaining case where $\mathcal{T} \not\succ T$ and $T \notin \mathcal{T}$. In this case, there is some $\sigma \in \mathcal{R}$ such that $T \subseteq \sigma$ and $\sigma \notin \rho_{\mathcal{R}}(\mathcal{T})$.

Case $|T| > 0$: Because $|\mathcal{T} \cup \{T\}| = d \leq \tau_1$ and every interaction in $\mathcal{T} \cup \{T\}$ has strength at least one, $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T} \cup \{T\})$ is not empty. Let T' be an interaction of strength t that appears in a test case in $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T} \cup \{T\})$ and has a different value on at least one

parameter from T. Let $\mathcal{T}' = \mathcal{T} \cup \{T'\}$. $\mathcal{T}'$ is independent because $\mathcal{T}$ is independent, and $\hat{T} \not\subset T'$ for any $\hat{T} \in \mathcal{T}$. Note that if $\hat{T} \subset T'$, $\hat{T}$ would occur in the test case with T'. Also, $\mathcal{T}' \not\succ T$ because $T \subseteq \sigma$, $\sigma \notin \rho_{\mathcal{R}}(\mathcal{T})$, and $\sigma \notin \rho_{\mathcal{R}}(T')$. Because A is a $(d, \overline{t})$-CDA, $\mathcal{T}'$ is independent, and $\mathcal{T}' \not\succ T$, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T}')$ holds. Hence, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$.

Case $T = \lambda$ (i.e., $|T| = 0$): As $\lambda \notin \mathcal{T}$, every interaction in $\mathcal{T}$ has strength at least one. Because $|\mathcal{T}| = d - 1 < \tau_1$, $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T})$ is not empty. Let T' be any t-way interaction that appears in a test case in $\mathcal{R} - \rho_{\mathcal{R}}(\mathcal{T})$. Let $\mathcal{T}' = \mathcal{T} \cup \{T'\}$. Because of the same argument as in the case $|T| > 0$, $\mathcal{T}'$ is independent and $\mathcal{T}' \not\succ T$, and thus $\rho_A(T) \not\subseteq \rho_A(\mathcal{T}')$. Hence, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$.

As a result, $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$. By induction, A is a $(d', \overline{t})$-CDA for any $0 \leq d' \leq d$, and thus is a $(\overline{d}, \overline{t})$-CDA. □

Theorem 8. *A $(\overline{d}, t)$-CDA is a t-CCA. A $(\overline{d}, \overline{t})$-CDA is a t-CCA.*

Proof. Let $T \in \mathcal{VI}_t$. Let A be a $(\overline{d}, t)$-CDA or $(\overline{d}, \overline{t})$-CDA. Then, $T \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$ for any $\mathcal{T} \subseteq \mathcal{VI}_t$ such that $|\mathcal{T}| \leq d$. If $|\mathcal{T}| = 0$, then $\mathcal{T} = \emptyset$, in which case $\mathcal{T} \not\succ T$, $T \notin \mathcal{T}$, and $\rho_A(\mathcal{T}) = \emptyset$. Hence, $\rho_A(T) \neq \emptyset$. □

Theorem 9. *$\mathcal{R}$, the exhaustive test suite, is a (d, t)-, $(d, \overline{t})$-, $(\overline{d}, t)$- and $(\overline{d}, \overline{t})$-CDA for any d and t.*

Proof. Let T be a valid interaction and $\mathcal{T}$ be a set of valid interactions. Below, the proof will show that $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_{\mathcal{R}}(T) \subseteq \rho_{\mathcal{R}}(\mathcal{T}))$. If $\mathcal{T} \not\succ T$ and $T \notin \mathcal{T}$, then there is some $\sigma \in \mathcal{R}$ such that $T \subseteq \sigma$ and $\forall T' \in \mathcal{T} : T' \not\subseteq \sigma$, in which case $\sigma \in \rho_{\mathcal{R}}(T) - \rho_{\mathcal{R}}(\mathcal{T})$. That is, $\mathcal{T} \not\succ T \Rightarrow (T \notin \mathcal{T} \Rightarrow \rho_{\mathcal{R}}(T) \not\subseteq \rho_{\mathcal{R}}(\mathcal{T}))$. In addition, $T \in \mathcal{T} \Rightarrow \rho_{\mathcal{R}}(T) \subseteq \rho_{\mathcal{R}}(\mathcal{T})$ holds trivially. As a result, the theorem follows. □

Theorem 10. *A (d, t)-CDA is also a (d, t)-CLA; a $(\overline{d}, t)$-CDA is also a $(\overline{d}, t)$-CLA; a $(d, \overline{t})$-CDA is also a $(d, \overline{t})$-CLA; and a $(\overline{d}, \overline{t})$-CDA is also a $(\overline{d}, \overline{t})$-CLA.*

Proof. Let A be a (d,t)-CDA. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be different sets of t-way interactions of size d and mutually distinguishable. $\rho_{\mathcal{R}}(\mathcal{T}_1) \neq \rho_{\mathcal{R}}(\mathcal{T}_2)$ by the definition of distinguishability. Without loss of generality, the proof assumes that $\mathcal{T}_1 \not\subset \mathcal{T}_2$. Denote the test case that exists in $\rho_{\mathcal{R}}(\mathcal{T}_1)$ but not in $\rho_{\mathcal{R}}(\mathcal{T}_2)$ as $\boldsymbol{\sigma}_e$. There exists at least one valid interaction T in $\mathcal{T}_1$ that is covered by $\boldsymbol{\sigma}_e$. $\mathcal{T}_2$ does not mask T, and $T \notin \mathcal{T}_2$ because $\boldsymbol{\sigma}_e \in \rho_{\mathcal{R}}(T) \wedge \boldsymbol{\sigma}_e \notin \rho_{\mathcal{R}}(\mathcal{T}_2)$. Because A is a (d,t)-CDA, $\mathcal{T}_2 \not\succ T$, and $T \notin \mathcal{T}_2$, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T}_2)$ holds. Hence, there exists a row $\boldsymbol{\sigma}'_e$ in A such that $\boldsymbol{\sigma}'_e$ covers T but does not cover any interactions in $\mathcal{T}_2$, that is, $\boldsymbol{\sigma}'_e \in \rho_A(\mathcal{T}_1) \wedge \boldsymbol{\sigma}'_e \notin \rho_A(\mathcal{T}_2)$ holds. Thus, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ holds; hence, A is a (d,t)-CLA. The same argument holds whenever $|\mathcal{T}_1|$ and $|\mathcal{T}_2|$ are at most d. Therefore, it follows that a $(\overline{d},t)$-CDA is a $(\overline{d},t)$-CLA.

Next, let A be a $(d,\overline{t})$-CDA or $(\overline{d},\overline{t})$-CDA. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be different sets consisting of exactly d interactions of strength at most t or at most d interactions of strength at most t, respectively. Then, the same argument for the case of (d,t)-CDAs and $(\overline{d},t)$-CDAs holds. As a result, the theorem follows. □

Theorem 11. *A $(d+t)$-CCA is a $(\overline{d},\overline{t})$-CDA.*

Proof. Suppose that A is a $(d+t)$-CCA. The theorem holds if $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$ for any $T \in \overline{\mathcal{VI}_t}$ and $\mathcal{T} \subseteq \overline{\mathcal{VI}_t}$ such that $0 \leq |\mathcal{T}| \leq d$, $T \notin \mathcal{T}$, $\mathcal{T} \not\succ T$, and $\{T\} \cup \mathcal{T}$ is independent. The proof shows this by constructing a valid interaction $\hat{T}$ of strength $\leq d+t$ that covers T but cannot appear with any interaction in $\mathcal{T}$ in the same row. If such a $\hat{T}$ exists, some row of A contains it because A is a $(t+d)$-CCA. This row is in $\rho_A(T)$ but not in $\rho_A(\mathcal{T})$; thus, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$.

Because $\mathcal{T} \not\succ T$, there must be a valid test case σ that covers T but does not cover any $T' \in \mathcal{T}$. Let $\sigma = \langle s_1, s_2, \ldots, s_k \rangle$. σ can be regarded as k-way interaction $\{(F_1, s_1), (F_2, s_2), \ldots, (F_k, s_k)\}$. $\hat{T}$ is constructed by starting from $\hat{T} = T$ and gradually expanding it by applying the following process for all $T' \in \mathcal{T}$. First, select any $(F_i, v) \in T'$ such that $s_i \neq v$. This can be done because T' is not covered by σ (and thus $\lambda \notin \mathcal{T}$). Then, add

(F_i, s_i) to $\hat{T}$. Finally, $\hat{T}$ becomes the desired interaction. □

Corollary 1. *A $(d+t)$-CCA is a $(\overline{d}, \overline{t})$-CLA.*

Proof. The corollary is obvious according to Theorems 10 and 11. □

CHAPTER 5

FAULT IDENTIFICATION

This chapter explains how CCA, CLA, and CDA arrays are used to detect and locate faulty interactions. The parameters of the arrays are considered as $d = 1$ and $t = 2$. Suppose that the 2-way CCA, $(1, 2)$-CLA, and $(1, 2)$-CDA shown in Figures 2.5a, 3.2, and 4.2 are used as test suites. Figures 5.1, 5.2, and 5.3 summarize the results of test cases when executed under the following two scenarios.

(1) The only faulty interaction is $T_\alpha = \{(F_1, 0), (F_2, 0)\}$.

(2) There are two faulty interactions $T_\beta = \{(F_1, 0), (F_3, 0)\}$ and $T_\gamma = \{(F_1, 0), (F_4, 1)\}$.

5.1 Case: Constrained Covering Arrays

In *Case* 1, within the test cases in the 2-CCA (Figure 2.5a), only $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ fail. The two-way interactions that appear only in those failed test cases are as follows (the faulty interaction is indicated by an underline):

$$\underline{\{(F_1, 0), (F_2, 0)\}} \qquad \{(F_1, 0), (F_3, 0)\} \qquad \{(F_1, 0), (F_4, 0)\}$$

$$\{(F_2, 0), (F_4, 0)\} \qquad \{(F_3, 0), (F_4, 0)\} \qquad \{(F_1, 0), (F_4, 3)\}$$

	F_1	F_2	F_3	F_4	Case1	Case2
$\boldsymbol{\sigma}_1$	0	0	0	0	**Fail**	**Fail**
$\boldsymbol{\sigma}_2$	0	0	0	3	**Fail**	**Fail**
$\boldsymbol{\sigma}_3$	0	1	1	1	Pass	**Fail**
$\boldsymbol{\sigma}_4$	0	1	2	2	Pass	Pass
$\boldsymbol{\sigma}_5$	1	0	0	2	Pass	Pass
$\boldsymbol{\sigma}_6$	1	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_7$	1	0	2	1	Pass	Pass
$\boldsymbol{\sigma}_8$	1	1	1	0	Pass	Pass
$\boldsymbol{\sigma}_9$	2	0	0	1	Pass	Pass
$\boldsymbol{\sigma}_{10}$	2	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_{11}$	2	0	1	2	Pass	Pass
$\boldsymbol{\sigma}_{12}$	2	1	2	0	Pass	Pass

Figure 5.1: 2-CCA and test outcomes in Cases 1 and 2.

In *Case* 2, the failed test cases are σ_1, σ_2, and σ_3; thus, the candidates for faulty interactions are

$$\{(F_1,0),(F_2,0)\} \qquad \underline{\{(F_1,0),(F_3,0)\}} \qquad \{(F_1,0),(F_4,0)\}$$
$$\{(F_2,0),(F_4,0)\} \qquad \{(F_3,0),(F_4,0)\} \qquad \{(F_1,0),(F_4,3)\}$$
$$\{(F_1,0),(F_3,1)\} \qquad \underline{\{(F_1,0),(F_4,1)\}} \qquad \{(F_2,1),(F_4,1)\}$$
$$\{(F_3,1),(F_4,1)\}$$

For both cases, it is impossible to further reduce the candidates of faulty interactions. It can be concluded that CCAs can be used to detect the existence of faulty t-way interactions. However, it is impractical for CCAs to identify the faulty interactions.

5.2 Case: Constrained Locating Arrays

Suppose that the $(1,2)$-CLA ($\overline{1},2$-CLA) shown in Figure 3.2 is used. In *Case* 1, the test cases $\boldsymbol{\sigma}_1$, $\boldsymbol{\sigma}_2$, and $\boldsymbol{\sigma}_3$ fail and all the other test cases pass. The interactions that appear only in the failed test cases are as follows:

$$\underline{\{(F_1,0),(F_2,0)\}} \qquad \{(F_1,0),(F_3,0)\} \qquad \{(F_3,0),(F_4,0)\}$$
$$\{(F_1,0),(F_4,3)\} \qquad \{(F_1,0),(F_4,1)\}$$

The core idea of CLAs is that they allow a test outcome to be uniquely associated with a set of faulty interactions, which is mathematically represented as $\mathcal{T}_1 = \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2)$. In this case, $\rho_A(\mathcal{T}) = \rho_A(\{T_\alpha\}) = \{\sigma_1,\sigma_2,\sigma_3\}$ holds only for $\mathcal{T} = \{\{(F_1,0),(F_2,0)\}\}$, provided that $\mathcal{T} \subseteq \mathcal{VI}_2$ and $|\mathcal{T}| \leq 1$. Thus, the faulty interaction is correctly identified.

Now consider *Case* 2. The failed test cases are the same as in *Case* 1, i.e., $\boldsymbol{\sigma}_1$, $\boldsymbol{\sigma}_2$, and $\boldsymbol{\sigma}_3$. Hence, the conclusion that T_α is the only faulty interaction is also the same. This incorrect result is caused by the number of faulty interactions not agreeing with the assumption (namely, $d = 1$). In general, if faulty interactions exceed the number assumed, CLAs may identify non-faulty interactions as faulty but also identify faulty interactions

	F_1	F_2	F_3	F_4	Case1	Case2
$\boldsymbol{\sigma}_1$	0	0	0	0	**Fail**	**Fail**
$\boldsymbol{\sigma}_2$	0	0	0	3	**Fail**	**Fail**
$\boldsymbol{\sigma}_3$	0	0	1	1	**Fail**	**Fail**
$\boldsymbol{\sigma}_4$	0	1	1	2	Pass	Pass
$\boldsymbol{\sigma}_5$	0	1	2	0	Pass	Pass
$\boldsymbol{\sigma}_6$	1	0	0	2	Pass	Pass
$\boldsymbol{\sigma}_7$	1	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_8$	1	0	1	2	Pass	Pass
$\boldsymbol{\sigma}_9$	1	1	1	1	Pass	Pass
$\boldsymbol{\sigma}_{10}$	1	1	2	0	Pass	Pass
$\boldsymbol{\sigma}_{11}$	1	1	2	2	Pass	Pass
$\boldsymbol{\sigma}_{12}$	2	0	0	1	Pass	Pass
$\boldsymbol{\sigma}_{13}$	2	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_{14}$	2	0	2	0	Pass	Pass
$\boldsymbol{\sigma}_{15}$	2	1	1	0	Pass	Pass
$\boldsymbol{\sigma}_{16}$	2	1	1	2	Pass	Pass
$\boldsymbol{\sigma}_{17}$	2	1	2	1	Pass	Pass

Figure 5.2: $(\overline{1}, 2)$-CLA and test outcomes in Cases 1 and 2.

as non-faulty.

5.3 Case: Constrained Detecting Arrays

Suppose that the $(1,2)$-CDA ($(\overline{1},2)$-CDA) shown in Figure 4.2 is used to locate faulty interactions. For *Case* 1, the failed test cases are $\boldsymbol{\sigma}_1$, $\boldsymbol{\sigma}_2$, $\boldsymbol{\sigma}_3$, $\boldsymbol{\sigma}_4$, and $\boldsymbol{\sigma}_5$. The interactions occurring only in the failed test cases are all identified as faulty. In this case, these interactions are

$$\underline{\{(F_1,0),(F_2,0)\}} \qquad \{(F_1,0),(F_3,0)\} \qquad \{(F_1,0),(F_4,3)\}$$

T_α is correctly identified as faulty, whereas $\{(F_1,0),(F_3,0)\}$ and $\{(F_1,0),(F_4,3)\}$ are incorrectly identified as faulty. Because $\{T_a\}$ masks $\{(F_1,0),(F_3,0)\}$ and $\{(F_1,0),(F_4,3)\}$ (i.e., $\{T_\alpha\} \succ \{(F_1,0),(F_3,0)\}$ and $\{T_\alpha\} \succ \{(F_1,0),(F_4,3)\}$), it is inherently impossible to determine that $\{(F_1,0),(F_3,0)\}$ and $\{(F_1,0),(F_4,3)\}$ are not faulty when T_α is faulty. However, it should be noted that if the assumption that the number of faulty interactions is $d = 1$ is relied on, just as in the case of the CLA above, one could correctly identify only T_α as faulty. In fact, Theorem 10 shows that any CDA is a CLA.

For *Case* 2, the failed test cases are $\boldsymbol{\sigma}_1$, $\boldsymbol{\sigma}_2$, $\boldsymbol{\sigma}_3$, $\boldsymbol{\sigma}_4$, and $\boldsymbol{\sigma}_8$. The interactions identified as faulty are

$$\underline{\{(F_1,0),(F_3,0)\}} \qquad \underline{\{(F_1,0),(F_4,1)\}} \qquad \{(F_1,0),(F_4,3)\}$$

Although the last interaction is in fact not faulty, all the faulty ones are correctly identified. In general, when using a CDA, faulty interactions are never wrongly identified as non-faulty, even if the number of faulty interactions exceeds the assumed number d.

	F_1	F_2	F_3	F_4	Case1	Case2
$\boldsymbol{\sigma}_1$	0	0	0	0	**Fail**	**Fail**
$\boldsymbol{\sigma}_2$	0	0	0	1	**Fail**	**Fail**
$\boldsymbol{\sigma}_3$	0	0	0	3	**Fail**	**Fail**
$\boldsymbol{\sigma}_4$	0	0	1	1	**Fail**	**Fail**
$\boldsymbol{\sigma}_5$	0	0	2	2	**Fail**	Pass
$\boldsymbol{\sigma}_6$	0	1	1	2	Pass	Pass
$\boldsymbol{\sigma}_7$	0	1	2	0	Pass	Pass
$\boldsymbol{\sigma}_8$	0	1	2	1	Pass	**Fail**
$\boldsymbol{\sigma}_9$	1	0	0	2	Pass	Pass
$\boldsymbol{\sigma}_{10}$	1	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_{11}$	1	0	1	0	Pass	Pass
$\boldsymbol{\sigma}_{12}$	1	0	2	1	Pass	Pass
$\boldsymbol{\sigma}_{13}$	1	1	1	1	Pass	Pass
$\boldsymbol{\sigma}_{14}$	1	1	2	0	Pass	Pass
$\boldsymbol{\sigma}_{15}$	1	1	2	2	Pass	Pass
$\boldsymbol{\sigma}_{16}$	2	0	0	0	Pass	Pass
$\boldsymbol{\sigma}_{17}$	2	0	0	1	Pass	Pass
$\boldsymbol{\sigma}_{18}$	2	0	0	2	Pass	Pass
$\boldsymbol{\sigma}_{19}$	2	0	0	3	Pass	Pass
$\boldsymbol{\sigma}_{20}$	2	0	1	2	Pass	Pass
$\boldsymbol{\sigma}_{21}$	2	0	2	0	Pass	Pass
$\boldsymbol{\sigma}_{22}$	2	1	1	0	Pass	Pass
$\boldsymbol{\sigma}_{23}$	2	1	1	1	Pass	Pass
$\boldsymbol{\sigma}_{24}$	2	1	2	2	Pass	Pass

Figure 5.3: $(\overline{1}, 2)$-CDA and test outcomes in Cases 1 and 2.

CHAPTER 6

GENERATION ALGORITHMS

This chapter demonstrates the algorithms designed to generate CLAs and CDAs. The target arrays are fixed to (d, t)-CLAs and (d, t)-CDAs for convenience of explanation. Algorithms for generating other CLAs and CDAs, i.e., $(\overline{d}, \overline{t})$-CLAs, $(\overline{d}, \overline{t})$-CDAs, etc., can be easily derived from the proposed algorithms. The remainder of the chapter is divided into two parts, one about generation algorithms for (d, t)-CLAs and the other about generation algorithms for (d, t)-CDAs. Both parts introduce two algorithms, each designed in different directions: one can generate minimum arrays given sufficient time, while the other can generate arrays in a short time.

6.1 Generation Algorithms for CLAs

In this section, two algorithms for generating (d, t)-CLAs are proposed. Although little research exists on the generation of LAs, there has already been a large body of research on CCA generation in the combinatorial interaction testing field. The two proposed algorithms are inspired by use of existing CCA generation algorithms.

6.1.1 Satisfiability-Based Algorithm

The first algorithm leverages a satisfiability solver. The problem of generating a CLA of a given size is reduced to the satisfiability problem of a logical expression. A logical expression is satisfiable iff it evaluates to true for some *valuation*, i.e., assignment of values to the variables. The algorithm first estimates the upper bound on the minimum size of a CLA and uses it as the initial CLA size. Then, it creates a logical expression that is satisfiable iff a CLA of the initial size exists. The logical expression is in turn evaluated by a satisfiability solver. In addition, the logical expression is specially designed so that the valuation that satisfies it directly represents a CLA. Satisfiability solvers can produce such a satisfying valuation when the expression is satisfiable; hence, a CLA can be obtained from the output of the solver. By repeating this process while decreasing the CLA size, the algorithm can obtain the smallest CLA.

Logic Expression

To represent an array with a collection of variables, the *naïve matrix model* is applied. The *naïve matrix model* was originally used by Hnich et al. [24] to find CAs. In this model, an $N \times k$ array is represented as an $N \times k$ matrix of integer variables as follows:

$$A = \begin{pmatrix} p_1^1 & \cdots & p_k^1 \\ \vdots & \ddots & \vdots \\ p_1^N & \cdots & p_k^N \end{pmatrix}$$

The variable p_i^n represents the value on the parameter F_i in the n-th test case. The domain of p_i^n is $S_i = \{0, 1, \ldots, |S_i| - 1\}$. For the array A to become a $(1, t)$-CLA, the following conditions are imposed on A using logical expressions:

(1) The rows of A represent valid test cases.

(2) All valid t-way interactions are tested.

(3) $\mathcal{T}_1 \not\sim \mathcal{T}_2 \Leftrightarrow \rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$, where $\mathcal{T}_1, \mathcal{T}_2$ are interaction sets containing one valid t-way interaction.

The logical expressions that represent the above two conditions are presented below. By conjuncting all the expressions, it is possible to obtain a single logical expression to check for satisfiability.

Condition 1 In A, the n-th row is expressed as a tuple of k variables $\langle p_1^n, p_2^n, \ldots, p_k^n \rangle$. As defined in Section 2, a test case is valid iff it satisfies the constraints represented by ϕ, which is a Boolean-valued formula over parameters $F_1, \ldots, F_k$. Let $\phi|_{p_1^n, p_2^n, \ldots, p_k^n}$ denote ϕ with each F_i replaced with p_i^n. Then, the following expression enforces A to only contain valid test cases:

$$\mathit{Valid} := \bigwedge_{n=1}^{N} \phi|_{p_1^n, p_2^n, \ldots, p_k^n}$$

Condition 2 The following condition ensures that all valid t-way interactions are covered by at least one row in the array A, where N is the size of rows of A:

$$\mathit{Cover}(\mathcal{VI}_t) := \bigwedge_{\{(p_{x_1}, v_{x_1}), \ldots, (p_{x_t}, v_{x_t})\} \in \mathcal{VI}_t} \bigvee_{n=1}^{N} \Big(\bigwedge_{i=1}^{t} (p_{x_i}^n = v_{x_j}) \Big)$$

Condition 3 To let the distinguishable pairs of interaction sets stay distinguishable in A, the following condition can be applied:

$$\mathit{Identify}_{CLA}(\mathcal{T}_a, \mathcal{T}_b) := \bigvee_{n=1}^{N} \Big(\bigvee_{a=1}^{d} \bigwedge_{j=1}^{t} (p_{a_j}^n = v_{a_j}) \oplus \bigvee_{b=1}^{d} \bigwedge_{k=1}^{t} (p_{b_k}^n = v_{b_k}) \Big)$$

where interaction sets $\mathcal{T}_a = \{\{F_{a_{1_1}}, v_{a_{1_1}}, \ldots, F_{a_{1_t}}, v_{a_{1_t}}\}, \ldots, \{F_{a_{d_1}}, v_{a_{d_1}}, \ldots, F_{a_{d_t}}, v_{a_{d_t}}\}\}$ and $\mathcal{T}_b = \{\{F_{b_{1_1}}, v_{b_{1_1}}, \ldots, F_{b_{1_t}}, v_{b_{1_t}}\}, \ldots, \{F_{b_{d_1}}, v_{b_{d_1}}, \ldots, F_{b_{d_t}}, v_{b_{d_t}}\}\}$ are indistinguishable, i.e., $\mathcal{T}_a \not\sim \mathcal{T}_b$.

This condition focuses on the distinguishable pairs of interaction sets. To keep pairs distinguishable in the array A, the condition requires at least one interaction in the one

set (for example, $\mathcal{T}_a$) to appear in a row in A and let no interactions in the other set ($\mathcal{T}_b$) appear in the same row. Under this condition, $\rho_A(\mathcal{T}_a) \neq \rho_A(\mathcal{T}_b)$ holds in the array A. Note that for given $\mathcal{T}_a$ and $\mathcal{T}_b$, $\rho_A(\mathcal{T}_a) \neq \rho_A(\mathcal{T}_b)$ holds iff $\mathit{Identify}_{CLA}(\mathcal{T}_a, \mathcal{T}_b)$ is satisfiable.

This dissertation defines $\mathcal{U}$ as follows:

$$\mathcal{U} := \{(\mathcal{T}_a, \mathcal{T}_b) \mid \mathcal{T}_a, \mathcal{T}_b \subseteq \mathcal{VI}_t, |\mathcal{T}_a|, |\mathcal{T}_b| = d, \mathcal{T}_a \not\sim \mathcal{T}_b\}$$

By ANDing $\mathit{Identify}_{CLA}(\mathcal{T}_a, \mathcal{T}_b)$ for all $(\mathcal{T}_a, \mathcal{T}_b) \in \mathcal{U}$, an expression that represents the third condition is obtained.

The whole expression The whole expression that will be checked for satisfiability is obtained by conjuncting the expressions defined above as follows:

$$\mathit{existCLA} := \mathit{Valid} \wedge \mathit{Cover}(\mathcal{VI}_t) \wedge \bigwedge_{(\mathcal{T}_a, \mathcal{T}_a) \in \mathcal{U}} \mathit{Identify}_{CLA}(\mathcal{T}_a, \mathcal{T}_b)$$

By checking the satisfiability of this expression, whether a (d, t)-CLA of size N exists or not can be determined. If it is satisfiable, then a CLA of size N exists. In this case, the satisfying valuation for the $N \times k$ variables p_i^n represents all the entries of one such CLA. Meanwhile, if the expression is unsatisfiable, then it can be concluded that no (d, t)-CLA of size N exists.

The satisfiability of the above expression can be checked using constraint satisfaction problem (CSP) solvers, satisfiability modulo theories (SMT) solvers, or Boolean satisfiability (SAT) solvers with a Boolean encoding of integers.

Computing $\mathcal{U}$

To construct the above logical expression *existCLA*, it is necessary to obtain $\mathcal{U}$ first (see the subscript of the $\bigwedge$ in the expression). Computing $\mathcal{U}$ requires $\mathcal{VI}_t$. The computation of $\mathcal{VI}_t$ is discussed later. Here, it is described how one can compute $\mathcal{U}$ when $\mathcal{VI}_t$ is available.

Consider enumerating all $\mathcal{T}_a$-$\mathcal{T}_b$ pairs such that $\mathcal{T}_a \not\sim \mathcal{T}_b$ and $|\mathcal{T}_a| = |\mathcal{T}_b| = d$. The problem here is how to decide whether or not a given $\mathcal{T}_a$ and $\mathcal{T}_b$ are distinguishable. This is solvable by using satisfiability solving. Let integer variables $p_1, p_2, \ldots, p_k$ symbolically represent a test case $\boldsymbol{\sigma}$; that is,

$$\boldsymbol{\sigma} = (p_1, p_2, \ldots, p_k)$$

The domain of p_i is $\{0, 1, \ldots, |S_i| - 1\}$. Note that S_i is the domain of parameter F_i.

By the definition of distinguishability, given such a $\mathcal{T}$-T pair, $\mathcal{T}_a$ and $\mathcal{T}_b$ are distinguishable iff the following condition holds:

$$\exists \boldsymbol{\sigma} \in \mathcal{R} : \exists T_a \in \mathcal{T}_a : T_a \subseteq \boldsymbol{\sigma} \wedge \forall T_b \in \mathcal{T}_b : T_b \not\subseteq \boldsymbol{\sigma} \vee$$
$$\exists T_b \in \mathcal{T}_b : T_b \subseteq \boldsymbol{\sigma} \wedge \forall T_a \in \mathcal{T}_a : T_a \not\subseteq \boldsymbol{\sigma}$$

In other words, the condition holds if there is a valid test case that covers at least one interaction in the interaction set $\mathcal{T}_a$ (or $\mathcal{T}_b$) but does not cover any interactions in the interaction set $\mathcal{T}_b$ (or $\mathcal{T}_a$). Hence, given $\mathcal{T}_a$ and $\mathcal{T}_b$, $\mathcal{T}_a \not\sim \mathcal{T}_b$ holds iff the following expression evaluates to true:

$$\textit{checkDistinguishable}(\mathcal{T}_a, \mathcal{T}_b) :=$$
$$\Big(\bigvee_{a=1}^{d}\bigwedge_{j=1}^{t}(p_{a_j} = v_{a_j}) \oplus \bigvee_{b=1}^{d}\bigwedge_{k=1}^{t}(p_{b_k} = v_{b_k})\Big) \wedge \phi|_{p_1,p_2,\ldots,p_k}$$

where $\mathcal{T}_a = \{\{F_{a_{1_1}}, v_{a_{1_1}}, \ldots, F_{a_{1_t}}, v_{a_{1_t}}\}, \ldots, \{F_{a_{d_1}}, v_{a_{d_1}}, \ldots, F_{a_{d_t}}, v_{a_{d_t}}\}\}$ and $\mathcal{T}_b = \{\{F_{b_{1_1}}, v_{b_{1_1}}, \ldots, F_{b_{1_t}}, v_{b_{1_t}}\}, \ldots, \{F_{b_{d_1}}, v_{b_{d_1}}, \ldots, F_{b_{d_t}}, v_{b_{d_t}}\}\}$.

$\mathcal{U}$ is obtained by, for every $\mathcal{T}_a, \mathcal{T}_b$ pair, checking the satisfiability of *checkDistinguishable* $(\mathcal{T}_a, \mathcal{T}_b)$ and keeping the pair in $\mathcal{U}$ if the expression is satisfiable.

The Algorithm

The CLA generation algorithm that uses satisfiability solving is shown as Algorithm 1. The algorithm repeatedly solves the problem of finding a (d, t)-CLA while varying the

array size N. The array size N starts with a value sufficiently large to ensure the existence of a CLA and is gradually decreased until no existence of a CLA of size N is proved. To obtain the initial value of N, the algorithm creates a $(d+t)$-CCA using an off-the-shelf algorithm (line 1), where the CCA generation algorithm is represented by the function *generateCCA($\mathcal{M}$,x)*, which returns an x-CCA. The algorithm uses the size of the CCA minus one as the initial N, as any $(d+t)$-CCA is a (d,t)-CLA. The $(d+t)$-CCA is also used for computing $\mathcal{VI}_t$ because all valid t-way interactions appear in the CCA. The algorithm enumerates all t-way interactions occurring in the array, thus obtaining $\mathcal{VI}_t$.

In the algorithm, *generateCLA($\mathcal{M}$, d, t, N, $\mathcal{U}$)* in line 8 represents a function that produces a (d,t)-CLA of size N by checking the satisfiability of the expression *existCLA*. If the expression is satisfiable, then the SMT solver returns the satisfying valuation, in which case a (d,t)-CLA of size N is obtained because the valuation represents the (d,t)-CLA. The size N is then decreased by one and the same process is repeated. If the result of the satisfiability check is UNSAT (unsatisfiable), no CLA of size N exists (denoted as $\perp$ in the algorithm). Then, the algorithm returns the CLA of size $N+1$ and terminates.

One might think that binary search could work better to vary N than the linear search adopted by the algorithm. In fact, this is not the case because showing unsatisfiability, that is, the nonexistence of a CLA, usually takes much longer time than showing satisfiability, that is, the existence of a CLA. The linear search delays solving an unsatisfiable expression until all possible sizes are checked, avoiding getting trapped in a long computation required for the unsatisfiable problem instance.

The size of the expression *existCLA* increases polynomially in k when $t, d, |S_i|$, and N are fixed. The expression can be expressed as a Boolean formula with a polynomial size increase because $|S_i|$ is fixed. The Boolean satisfiability problem (SAT) is NP-complete in general, and there is no reason that the SAT can be solved in polynomial-time for this particular case. Hence, the time complexity of the algorithm is likely to be exponential.

6.1.2 Heuristic Algorithm

Algorithm 2 is the second generation algorithm for (d, t)-CLAs. The algorithm takes an SUT model $\mathcal{M}$ and integers d, t as input and finally returns a (d, t)-CLA A. The algorithm employs Theorem 3. This theorem shows that a $(d + t)$-CCA is already a (d, t)-CLA and can identify d t-way faulty interactions. However, a large number of redundant test cases are included in the CCA, which directly increases the testing code. The algorithm finds and deletes these redundant test cases so that the testing cost of the CLAs can be reduced. This algorithm is a heuristic algorithm because it does not guarantee that the output CLA is optimal in size. Indeed, the resulting CLAs can vary for different runs.

In the first line of the algorithm, the function *generateCCA()* uses an existing algorithm to generate a $(d + t)$-CCA. Then, the function *getAllInteractions()* is called to enumerate all t-way interactions the $(d + t)$-CCA contains. The interactions obtained are the set of all valid t-way interactions (i.e., $\mathcal{VI}_t$) because all interactions occurring in a CCA are valid and any $(d + t)$-CCA contains all t-way valid interactions. Once all the valid t-way interactions have been collected, it computes a mapping $Rows[]$, which maps each of them to the set of rows of A that cover it; that is, $Rows[] : T \mapsto \rho_A(T)$, where $T \in \mathcal{VI}_t$. Then, using $Rows[]$, a new mapping that maps every interaction set to covering test case sets is constructed, i.e., $SetRows[] : \mathcal{T} \mapsto \rho_A(\mathcal{T})$. This is computed by the function *getSetRows()*.

In each iteration of the while loop, a row $\boldsymbol{\sigma}$ is randomly chosen from the CCA. Then, the algorithm computes $SetRows'$, which is an upgraded mapping such that $SetRows'[] : \mathcal{T} \mapsto \rho_A(\mathcal{T})\backslash\{\boldsymbol{\sigma}\}$. In other words, $SetRows'$ is $\rho_{A'}(T)$, where A' is the array obtained from A by removing $\boldsymbol{\sigma}$ from it. The function *update()* is used to obtain $SetRows'$.

In each iteration of the loop, it is checked whether $\boldsymbol{\sigma}$ can be removed or not. The row can be removed if A remains a (d, t)-CLA after the removal. This check is performed by checking two conditions.

One condition is that every valid t-way interaction T still has some row that covers it; i.e., $Rows'[T] \neq \emptyset$. It is necessary to check this condition because all t-way interactions must be checked by at least one test case.

The other condition corresponds to the case where $|\mathcal{T}_1|$ and $|\mathcal{T}_2|$ are distinguishable. The condition is that for every pair of valid, mutually distinguishable t-way interactions, they still have different sets of rows in which they are covered. In other words, for $T_a, T_b \in \mathcal{VI}_t$, if $\mathcal{T}_a$ and $\mathcal{T}_b$ are distinguishable, then $SetRows'(\mathcal{T}_a) \neq SetRows'(\mathcal{T}_b)$ (i.e., $\rho_{A'}(\mathcal{T}_a) \neq \rho_{A'}(\mathcal{T}_b)$).

Clearly, if an interaction T is not covered by $\boldsymbol{\sigma}$, the deletion of $\boldsymbol{\sigma}$ does not alter the set of rows that cover T. Hence, checking the two conditions can be performed by examining only the interactions covered by $\boldsymbol{\sigma}$, instead of all interactions in $\mathcal{VI}_t$.

The loop is iterated until all rows in the initial A have been examined. Finally, the resulting A becomes a (d, t)-CLA of reduced size.

As stated above, output (d, t)-CLAs vary for different runs of the algorithm, even if the initial A (i.e., the $(d + t)$-CCA generated in line 12) is identical for all runs. This is because the final (d, t)-CLAs obtained also depend on the order of row deletion.

Let $s = \max_{1 \leq i \leq k} |S_i|$. Outside the while loop, line 15 has the highest time complexity. It is $O((s^t k^t)^{2d} n)$ because $|\mathcal{VI}_t| \leq s^t k^t$, $|\rho_S()| \leq n$. In the algorithm, lines 20 and 21 have the highest complexity $O((s^t k^t)^{2d} n)$ for the same reason. Let n be the size of the initial CCA. As a result, the algorithm's time complexity is $O((s^t k^t)^{2d} n^2)$. When s, t, and d are fixed, the complexity is polynomial in k and n.

6.2 Generation Algorithms for CDAs

This section presents two algorithms for generating CDAs: the satisfiability-based algorithm and heuristic algorithm. The generation of CDAs is limited to (d, t)-CDAs because

(d, t)-CDAs are $(\overline{d}, t)$-CDAs except in extreme cases (Theorem 7). Also, it is straightforward to adjust the algorithms to $(d, \overline{t})$-CDAs and $(\overline{d}, \overline{t})$-CDAs.

6.2.1 Satisfiability-Based Algorithm

Similar to the satisfiability-based algorithm for CLAs, this algorithm also leverages a satisfiability solver. Because a $(d + t)$-CCA is already a (d, t)-CDA, the algorithm uses the size of a (d, t)-CCA as the initial size of a CDA. Then, it creates a logical expression that is satisfiable iff a CDA of the initial size exists. The logical expression is subsequently evaluated by a satisfiability solver. A CDA instance can be obtained from the output of the solver if the logical expression is satisfiable. By repeating this process while decreasing the CDA size, the algorithm can output the smallest CDA.

Logic Expression

The *naïve matrix model*, an $N \times k$ array, is used as a representation of an $N \times k$ matrix of integer variables as follows:

$$A = \begin{pmatrix} p_1^1 & \cdots & p_k^1 \\ \vdots & \ddots & \vdots \\ p_1^N & \cdots & p_k^N \end{pmatrix}$$

The variable p_i^n represents the value on the parameter F_i in the n-th test case. The domain of p_i^n is $S_i = \{0, 1, \ldots, |S_i| - 1\}$. For the array A to become a (d, t)-CDA, the following conditions are applied to A:

(1) The rows of A represent valid test cases.

(2) $\forall \mathcal{T} \subseteq \mathcal{VI}_t$ such that $|\mathcal{T}| = d, \forall T \in \mathcal{VI}_t : \mathcal{T} \not> T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$

The reason that the conditions do not require all valid t-way interactions to be tested is that the definition of (d, t)-CDA implies that all valid t-way interactions appear in at least

one row, i.e., $\forall T \in \mathcal{VI}_t, \rho_A(T) \neq \emptyset$ iff A is a (d,t)-CDA. The logical expressions that represent the above two conditions are presented below. A single logical expression can be obtained by conjuncting all the expressions and will be checked for satisfiability.

Condition 1 The following expression enforces A to only contain valid test cases:

$$\mathit{Valid} := \bigwedge_{n=1}^{N} \phi|_{p_1^n, p_2^n, \ldots, p_k^n}$$

Condition 2 It is important to note that $\mathcal{T} \not\succ T \Rightarrow (T \in \mathcal{T} \Leftrightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T}))$ is equivalent to

$$(\mathcal{T} \not\succ T \wedge T \notin \mathcal{T}) \Rightarrow \rho_A(T) \not\subseteq \rho_A(\mathcal{T})$$

because $T \in \mathcal{T} \Rightarrow \rho_A(T) \subseteq \rho_A(\mathcal{T})$ holds trivially. Hence, it is able to focus on the case where $\mathcal{T} \not\succ T$ and $T \notin \mathcal{T}$. The right part of this formula, that is, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$, holds iff there is a row in A that covers T but none of the interactions in $\mathcal{T}$. This condition is represented by a logical expression as follows:

$$\mathit{Identify}_{CDA}(\mathcal{T}, T) := \bigvee_{n=1}^{N} \Big(\bigwedge_{j=1}^{t} (p_{x_j}^n = v_{x_j}) \wedge \neg \Big(\bigvee_{L=1}^{d} \bigwedge_{l=1}^{t} (p_{y_{L_l}}^n = v_{y_{L_l}}) \Big) \Big)$$

where $\mathcal{T} = \{\{(F_{y_{1_1}}, v_{y_{1_1}}), \ldots, (F_{y_{1_t}}, v_{y_{1_t}})\}, \ldots, \{(F_{y_{d_1}}, v_{y_{d_1}}), \ldots, (F_{y_{d_t}}, v_{y_{d_t}})\}\}$ and $T = \{(F_{x_1}, v_{x_1}), \ldots, (F_{x_t}, v_{x_t})\}$. For given $\mathcal{T}$ and T, $\rho_A(T) \not\subseteq \rho_A(\mathcal{T})$ holds iff *Locating*$(\mathcal{T}, T)$ is satisfiable.

Then, $\mathcal{V}$ is defined as follows:

$$\mathcal{V} := \{(\mathcal{T}, T) \mid \mathcal{T} \subseteq \mathcal{VI}_t, |\mathcal{T}| = d, T \in \mathcal{VI}_t, \mathcal{T} \not\succ T, T \notin \mathcal{T}\}$$

By ANDing *Locating*$(\mathcal{T}, T)$ for all $(\mathcal{T}, T) \in \mathcal{V}$, an expression that represents the second condition can be constructed.

The whole expression The whole expression that will be checked for satisfiability is obtained by conjuncting the expressions defined above as follows:

$$\textit{existCDA} := \textit{Valid} \wedge \bigwedge_{(\mathcal{T},T)\in\mathcal{V}} \textit{Locating}(\mathcal{T}, T)$$

By checking the satisfiability of this expression, whether a (d, t)-CDA of size N exists or not can be determined. If it is satisfiable, then a CDA of size N exists. In this case, the satisfying valuation for the $N \times k$ variables p_i^n represents all the entries of one such CDA. Meanwhile, if the expression is unsatisfiable, then it can be concluded that no (d, t)-CDA of size N exists.

Computing $\mathcal{V}$

Before constructing the expression *existCDA*, the expression $\mathcal{V}$ must be examined first.

Consider enumerating all $\mathcal{T}$-T pairs such that $T \in \mathcal{VI}_t$, $\mathcal{T} \subseteq \mathcal{VI}_t$, $|\mathcal{T}| = d$, $T \notin \mathcal{T}$, and $\mathcal{T} \not\succ T$. The problem here is how to decide whether or not $\mathcal{T}(\subseteq \mathcal{VI}_t)$ masks $T(\in \mathcal{VI}_t)$ when $\mathcal{T}$ and $T \notin \mathcal{T}$ are given. This too is possible by employing satisfiability solving. Let integer variables $p_1, p_2, \ldots, p_k$ symbolically represent a test case $\boldsymbol{\sigma}$; that is,

$$\boldsymbol{\sigma} = (p_1, p_2, \ldots, p_k)$$

The domain of p_i is $\{0, 1, \ldots, |S_i| - 1\}$. Note that S_i is the domain of parameter F_i.

By the definition of masking, given such a $\mathcal{T}$-T pair, $\mathcal{T}$ does not mask T iff the following condition holds:

$$\exists \boldsymbol{\sigma} \in \mathcal{R} : T \subseteq \boldsymbol{\sigma} \wedge \neg(\exists T' \in \mathcal{T} : T' \subseteq \boldsymbol{\sigma})$$

In other words, the condition holds if there is a valid test case that covers the interaction T but does not cover any interactions in the interaction set $\mathcal{T}$. Hence, given $\mathcal{T}$ and T,

$\mathcal{T} \not\succ T$ holds iff the following expression evaluates to true:

$$\textit{checkUnMasking}(\mathcal{T}, T) :=$$
$$\bigwedge_{j=1}^{t} (p_{x_j} = v_{x_j}) \wedge \neg\Big(\bigvee_{L=1}^{d} \bigwedge_{l=1}^{t} (p_{y_{L_l}} = v_{y_{L_l}}) \Big) \wedge \phi|_{p_1,p_2,\ldots,p_k}$$

where $\mathcal{T} = \{\{(F_{y_{1_1}}, v_{y_{1_1}}), \ldots, (F_{y_{1_t}}, v_{y_{1_t}})\}, \ldots, \{(F_{y_{d_1}}, v_{y_{d_1}}), \ldots, (F_{y_{d_t}}, v_{y_{d_t}})\}\}$ and $T = \{(F_{x_1}, v_{x_1}), \ldots, (F_{x_t}, v_{x_t})\}$.

$\mathcal{U}$ is obtained by, for every $\mathcal{T}$-T pair, checking the satisfiability of *checkUnMasking*$(\mathcal{T}, T)$ and keeping the pair in $\mathcal{U}$ if the expression is satisfiable.

The Algorithm

The satisfiability-based generation algorithm for CDAs is shown as Algorithm 3. The algorithm repeatedly solves the problem of finding a (d, t)-CDA while reducing the size from initial N to the size of an optimal CDA. The initial size N is a value large enough that the existence of a CDA is ensured. By decreasing N, the algorithm finally reaches a size for which no CDA exists. Finally, the last constructed CDA will be proved to be the optimal CDA.

To obtain the initial value of N, the algorithm creates a $(d + t)$-CCA using an off-the-shelf algorithm (line 29), where the CCA generation algorithm is represented as the function *generateCCA(*$\mathcal{M}$*,x)*, which returns an x-CCA. Then, the algorithm uses the size of the CCA minus one as the initial N, as any $(d + t)$-CCA is a (d, t)-CDA. The $(d + t)$-CCA is also used for computing $\mathcal{VI}_t$ because all valid t-way interactions appear in the CCA.

In the algorithm, *generateCDA(*$\mathcal{M}$*, d, t, N,* $\mathcal{U}$*)* in line 36 represents a function that produces a (d, t)-CDA of size N by checking the satisfiability of the expression *existCDA*. If the expression is satisfiable, then the SMT solver returns the satisfying valuation, in which case a (d, t)-CDA of size N is obtained because the valuation represents the (d, t)-

CDA. Then, the size N is decreased by one, and the same process is repeated. If the result is UNSAT (unsatisfiable), no CDA of size N exists (denoted as $\perp$ in the algorithm). Then, the algorithm returns the CDA of size $N + 1$ and terminates.

The time complexity of the algorithm is likely to be exponential for the same reason explained in the previous section.

6.2.2 Heuristic Algorithm

In this section, a heuristic algorithm is proposed for the generation of (d, t)-CDAs, which aims to generate (d, t)-CDAs that are not optimal but fairly small in reasonable time.

Theorem 11 shows that a $(d+t)$-CCA is already a (d, t)-CDA. Based on this theorem, a heuristic algorithm (Algorithm 4) can be devised. The algorithm generates a $(d + t)$-CCA first. Then, it repeatedly chooses a test case in it at random and checks whether it is removable. A test case is judged as removable from an array if a new array with the test case being removed would still be a (d, t)-CDA. If the test case is removable, then it is removed from the current array. Otherwise, a new test case is chosen and the check is performed again. This process is repeated until no test case is removable anymore.

The details of the algorithm are as follows. In line 40, the algorithm generates a $(d+t)$-CCA S. At this point, S is already a (d, t)-CDA but contains many redundant test cases. Then, the algorithm collects all valid t-way interactions and maps each interaction T to its covering test cases $\rho_S(T)$ in S (line 41). The map obtained here, denoted by $Rows[]$, is used to compute another map, *DiffRows*[][], which associates each pair of an interaction set $\mathcal{T}$ and valid interaction T with $\rho_S(T) - \rho_S(\mathcal{T})$. Note that $\rho_S(T) - \rho_S(\mathcal{T}) = \emptyset$ iff $\rho_S(T) \subseteq \rho_S(\mathcal{T})$. Because S is a CDA, *DiffRows*$[\mathcal{T}][T] = \emptyset$ if $\mathcal{T} \succ T$, and *DiffRows*$[\mathcal{T}][T] \neq \emptyset$ otherwise.

Then, the algorithm repeatedly chooses a test case at random and checks whether it is removable or not. To perform the check, the algorithm constructs a new interaction-

to-row map *DiffRows*$'[][]$ that would hold after the test case was removed (line 49). This can be done by simply removing σ from all *DiffRows*$[\mathcal{T}][T]$. Subsequently, the algorithm compares the two maps (line 50). If *DiffRows*$[\mathcal{T}][T] \neq \emptyset$ but *DiffRows*$'[\mathcal{T}][T] = \emptyset$, then $\rho_S(T) \subseteq \rho_S(\mathcal{T})$, and thus S is no longer a CDA. In this case, the algorithm reserves the test case for the output test suite A (line 51). Otherwise, it deletes the test case and accordingly updates *DiffRows*$[\mathcal{T}][T]$ (line 54). When all test cases in the CCA are checked, the algorithm will terminate, yielding the resulting A.

Let $s = \max_{1 \leq i \leq k} |S_i|$. Outside the while loop, line 43 has the highest time complexity, which is $O((s^t k^t)^d s^t k^t n)$. Inside the while loop, for the same reason, lines 49 and 50 has the highest complexity of $O((s^t k^t)^d s^t k^t n)$. Let n be the size of the initial CCA. The algorithm's time complexity is $O((s^t k^t)^d s^t k^t n^2)$. When s, t, and d are fixed, the complexity is polynomial in k and n.

Snippet 1: CLA: satisfiability-based algorithm

Input: SUT $\mathcal{M} = \langle \mathcal{F}, \mathcal{S}, \phi \rangle$; integers d, t

Output: (d,t)-CLA A

// construct a $(d+t)$-CCA for the input SUT

1 $S \leftarrow$*generateCCA*($\mathcal{M}$, $d+t$)

// get all valid t-way interactions from the $(d+t)$-CCA

2 $\mathcal{VI}_t \leftarrow$ *getAllInteractions*(S, t)

// get all distinguishable pairs $\mathcal{U}$ of interaction sets

3 $\mathcal{U} \leftarrow getU(\mathcal{VI}_t, d, t)$

// get the initial size for the CLA to be generated

4 $N \leftarrow$ The size of $S - 1$

5 $nextA \leftarrow S$

6 **do**

  // reserve the current test suite instance

7 $A \leftarrow nextA$

  // SAT checking; the solver returns an instance if satisfiable or the emptyset otherwise

8 $nextA \leftarrow$*generateCLA*($\mathcal{M}, d, t, N, \mathcal{U}$)

  // decrease the size by one

9 $N \leftarrow N - 1$

10 **while** $nextA \neq \perp$

11 **return** A

Snippet 2: CLA: heuristic algorithm

Input: SUT $\mathcal{M} = \langle \mathcal{F}, \mathcal{S}, \phi \rangle$; integers d, t

Output: (d,t)-CLA A

// construct a $(d+t)$-CCA for the input SUT

12 $S \leftarrow$*generateCCA*($\mathcal{M}$, $d+t$)

// get all t-way interactions from the $(d+t)$-CCA

13 $\mathcal{VI}_t \leftarrow$ *getAllInteractions*(S, t)

// $Rows[T] = \rho_S(T)$ for $T \in \mathcal{VI}_t$

14 *Rows*[] $\leftarrow$ *mapInteractionToRows*$(\mathcal{VI}_t, S)$

// *SetRows*$[\mathcal{T}] = \cup_{T \in \mathcal{T}} \rho_S(T)$ for $\mathcal{T} \subseteq \mathcal{VI}_t, |\mathcal{T}| = d$

15 *SetRows*[] $\leftarrow$ *getSetRows*$(\mathcal{VI}_t, S, d)$

16 $A \leftarrow S$

17 **while** $S \neq \emptyset$ **do**

18 $\quad \boldsymbol{\sigma} \leftarrow$ *getRandomTestcase*(S)

19 $\quad S \leftarrow S - \{\boldsymbol{\sigma}\}$; $A \leftarrow A - \{\boldsymbol{\sigma}\}$

20 $\quad$ *SetRows*$'$[] $\leftarrow$ *update*(*SetRows*[], $\boldsymbol{\sigma}$); *Rows*$'$[] $\leftarrow$ *update*(*Rows*[], $\boldsymbol{\sigma}$)

21 $\quad$ **if** $\exists \mathcal{T}_a, \mathcal{T}_b$ *s.t.* $\mathcal{T}_a \not\sim \mathcal{T}_b$: *SetRows*$'[\mathcal{T}_a] =$ *SetRows*$'[\mathcal{T}_b]$ *or* $\exists T$: *Rows*$'[T] = \emptyset$ **then**

$\quad\quad$ // the test case $\boldsymbol{\sigma}$ is unremovable

22 $\quad\quad A \leftarrow A + \{\boldsymbol{\sigma}\}$

23 $\quad$ **end**

24 $\quad$ **else**

$\quad\quad$ // the test case $\boldsymbol{\sigma}$ is removable

25 $\quad\quad$ *SetRows*[] $\leftarrow$ *SetRows*$'$[]

26 $\quad$ **end**

27 **end**

28 **return** A

Snippet 3: CDA: satisfiability-based algorithm

Input: SUT $\mathcal{M} = \langle \mathcal{F}, \mathcal{S}, \phi \rangle$; integers d, t

Output: (d,t)-CDA A

// construct a $(d+t)$-CCA for the input SUT

29 $S \leftarrow$*generateCCA*($\mathcal{M}$, $d+t$)

// get all valid t-way interactions from the $(d+t)$-CCA

30 $\mathcal{VI}_t \leftarrow$ *getAllInteractions*(S, t)

// get all non-masking pairs $\mathcal{U}$ of interaction sets and interactions

31 $\mathcal{U} \leftarrow getU(\mathcal{VI}_t, d, t)$

// get the initial size for the CDA to be generated

32 $N \leftarrow$ The size of $S - 1$

33 $nextA \leftarrow S$

34 **do**

  // reserve the current test suite instance

35 $A \leftarrow nextA$

  // SAT checking; the solver returns an instance if satisfiable or the emptyset, otherwise

36 $nextA \leftarrow$*generateCDA*($\mathcal{M}, d, t, N, \mathcal{U}$)

  // decrease the size by one

37 $N \leftarrow N - 1$

38 **while** $nextA \neq \perp$

39 **return** A

Snippet 4: CDA: heuristic algorithm

Input: SUT $\mathcal{M} = \langle \mathcal{F}, \mathcal{S}, \phi \rangle$; integers d, t

Output: (d, t)-CDA A

// construct a $(d + t)$-CCA for the input SUT

40 $S \leftarrow$ *generateCCA*$(\mathcal{M}, d + t)$

// get all t-way interactions from the $(d + t)$-CCA

41 $\mathcal{VI}_t \leftarrow$ *getAllInteractions*(S, t)

// $Rows[T] = \rho_S(T)$ for $T \in VI_t$

42 *Rows*[] $\leftarrow$ *mapInteractionToRows*$(\mathcal{VI}_t, S)$

// *DiffRows*$[\mathcal{T}][T] = \rho_S(T) - \rho_S(\mathcal{T})$ for $\mathcal{T} \subseteq \mathcal{VI}_t, |\mathcal{T}| = d$

43 *DiffRows*[][] $\leftarrow$ *getDiffRows*$(\mathcal{VI}_t, S, d)$

44 $A \leftarrow S$

45 **while** $S \neq \emptyset$ **do**

46 $\quad \boldsymbol{\sigma} \leftarrow$ *getRandomTestcase*(S)

47 $\quad S \leftarrow S - \{\boldsymbol{\sigma}\}$

48 $\quad A \leftarrow A - \{\boldsymbol{\sigma}\}$

49 $\quad$ *DiffRows*$'$[][] $\leftarrow$ *update*(*DiffRows*[][], $\boldsymbol{\sigma}$)

50 $\quad$ **if** $\exists \mathcal{T}, T$: *DiffRows*$[\mathcal{T}][T] \neq \emptyset$ *and* *DiffRows*$'[\mathcal{T}][T] = \emptyset$ **then**

$\quad\quad$ // the test case $\boldsymbol{\sigma}$ is unremovable

51 $\quad\quad A \leftarrow A + \{\boldsymbol{\sigma}\}$

52 $\quad$ **end**

53 $\quad$ **else**

$\quad\quad$ // the test case $\boldsymbol{\sigma}$ is removable

54 $\quad\quad$ *DiffRows*[][] $\leftarrow$ *DiffRows*$'$[][]

55 $\quad$ **end**

56 **end**

57 **return** A

CHAPTER 7

EXPERIMENTS

7.1 Experiment Purposes and Research Questions

This chapter shows experimental results on the proposed arrays and generation algorithms. The array generation is mainly focused on $(1, 2)$-CLAs and $(1, 2)$-CDAs ($d = 1$ and $t = 2$) for the following reasons. First, it is natural to set d to a small value in practice. Second, the most common form of CIT targets two-way interactions.

In the following sections, the experimental results on the generation algorithms are presented first. Two different experiments were conducted with respect to two different purposes of analysis. One was comparing the efficiency between two different algorithm types, i.e., satisfiability-based and heuristic algorithms, for both CLAs and CDAs. The other was analyzing the efficiency of the heuristic algorithms when the strength t is greater than 2. After discussing the generation algorithms, fault identification using CLAs and CDAs for real-world systems is reported.

To guide the experiments, research questions were set as follows:

RQ1 How do the two types of generation algorithms perform with respect to generation time and array size?

RQ2 How do the heuristic algorithms perform when the strengths t of CLAs and CDAs are relatively large ($t \geq 2$)?

RQ3 Can CLAs and CDAs be used to identify faulty interactions caused by actual bugs, especially when the assumptions about the number and strength of faulty interactions do not hold?

7.2 Comparison of Generation Algorithms

7.2.1 Experimental Settings

The generation algorithms were written in C++ language. The CCA generator [61] used in the programs is an implementation of the IPOG algorithm [38], while the Z3 solver (version 4.8.1) [15] was used as the satisfiability checker in the satisfiability-based algorithms.

All experiments were conducted on a machine with an Intel Core i7-8700 CPU, 64 GB memory, and Ubuntu 18.04 LTS OS. For each benchmark instance, the heuristic algorithms were executed 5 times, as they are non-deterministic algorithms. The satisfiability-based algorithms were run only once because they are deterministic. The timeout period for each run was set to 1800s.

A total of 20 benchmark instances, numbered from 1 to 20, were performed in the experiments. These benchmarks can be found in [53]. Detailed information about these benchmark instances is listed in Table 7.1. In the table, the first two columns show the benchmark ID and benchmark names. Columns $|\mathcal{F}|$ and $|\phi|$ show the numbers of parameters and constraints in the benchmarks, respectively. Then, the columns $|\mathcal{VI}_2|$ and $|\mathcal{I}_2 \backslash \mathcal{VI}_2|$ show the numbers of valid and invalid 2-way interactions. The column with the label $|\mathcal{T}_a \sim \mathcal{T}_b|$ shows the numbers of indistinguishable pairs of interaction sets. The final column with the label $|\mathcal{T} \succ T|$ shows the numbers of masking pairs of interaction sets

Table 7.1: Benchmark information

ID	SUT	$\lvert\mathcal{F}\rvert$	$\lvert\phi\rvert$	$\lvert\mathcal{VI}_2\rvert$	$\lvert\mathcal{I}_2 \backslash \mathcal{VI}_2\rvert$	$\lvert\mathcal{T}_a \sim \mathcal{T}_b\rvert$	$\lvert\mathcal{T} \succ T\rvert$
1	banking1	5	112	102	0	0	0
2	banking2	15	3	473	3	0	208
3	comm_protocol	11	128	285	35	16	2,177
4	concurrency	5	7	36	4	69	130
5	healthcare1	10	21	361	8	5	512
6	healthcare2	12	25	466	1	0	124
7	healthcare3	29	31	3,092	59	477	8,700
8	healthcare4	35	22	5,707	38	288	3,359
9	insurance	14	0	4,573	0	0	0
10	network_mgmt	9	20	1,228	20	0	189
11	processor_comm1	15	13	1,058	13	6	1,510
12	processor_comm2	25	125	2,525	854	1,562	35,156
13	services	13	388	1,819	16	93	1,088
14	storage1	4	95	53	18	11	112
15	storage2	5	0	126	0	0	0
16	storage3	15	48	1,020	120	57	3,400
17	storage4	20	24	3,491	24	0	0
18	storage5	23	151	5,342	246	20	10,095
19	system_mgmt	10	17	310	14	130	825
20	telecom	10	21	440	11	23	151

and valid interactions. Note that the numbers of indistinguishable interaction sets and of masking pairs are the values for the case of $d = 1$ and $t = 2$.

7.2.2 CLA: Experimental Results

These experimental results are listed in Table 7.2. The first column shows the benchmark ID. The values are divided into two sections: generation time and sizes of generated CLAs. In each section, the average value is reported for the satisfiability-based algorithm, as it is the deterministic, while the maximum, minimum, and average values are reported for the heuristic algorithm. The symbol "– –" is used to indicate the case that the proposed algorithms did not terminate within the timeout period. To better compare the generation results, shorter generation time and smaller sizes of CLAs are noted in bold font.

The satisfiability-based algorithm completed the generation process for three instances, namely, Nos. 1, 4, and 14. The CLAs obtained for these instances are optimal. However, the algorithm failed to generate even a single CLA for the other instances. In contrast, the heuristic algorithm successfully generated CLAs for all benchmark instances. In addition, the execution time of the satisfiability-based algorithm was always much longer than that of the other algorithm, sometimes four orders of magnitude longer. There are two main reasons why the satisfiability-based algorithm is so slow. One is that the algorithm generates multiple CLAs in a single run. As stated in Chapter 6, it generates (d, t)-CLAs with sizes varying from the size of a $(d + t)$-CCA. The CCA's size simply serves as the upper bound on the minimum CLA size. As this is not tight bound in general, to obtain an optimal (d, t)-CLA, the satisfiability solver is executed multiple times. The other reason, which is more obvious, is that the satisfiability check may be time-consuming. The time required for the check becomes very long, especially when the algorithm tries to find a CLA of minimum size minus 1, in which case the answer of the check is UNSAT (unsatisfiable). In the field of satisfiability, it is well known that UNSAT instances are usually

Table 7.2: CLA: Experimental results of generating CLAs

ID	Time (second)				Size			
	SMT	Two-step			SMT	Two-step		
	value	max.	min.	avg.	value	max.	min.	avg.
1	626.05	0.11	0.10	**0.10**	**24**	28	26	26.8
2	– –	0.22	0.15	**0.19**	– –	29	27	**28**
3	– –	0.20	0.16	**0.17**	– –	35	33	**34.40**
4	0.53	0.08	0.07	**0.07**	**7**	7	7	**7**
5	– –	0.22	0.20	**0.21**	– –	49	46	**47.60**
6	– –	0.27	0.23	**0.24**	– –	36	33	**34.40**
7	– –	8.59	7.91	**8.20**	– –	95	84	**91**
8	– –	45.07	43.51	**44.32**	– –	107	102	**104.20**
9	– –	124.91	123.87	**124.39**	– –	803	794	**798.40**
10	– –	2.60	2.56	**2.58**	– –	210	202	**206.40**
11	– –	1.03	0.85	**0.92**	– –	61	57	**59.20**
12	– –	3.48	3.31	**3.40**	– –	70	65	**67.40**
13	– –	7.49	7.28	**7.41**	– –	203	193	**198.40**
14	7.57	0.22	0.08	**0.11**	**22**	22	22	**22**
15	– –	0.09	0.09	**0.09**	– –	36	35	**35.60**
16	– –	1.17	1.11	**1.15**	– –	91	85	**88.60**
17	– –	30.28	29.28	**29.72**	– –	222	215	**218**
18	– –	112.85	112.09	**112.48**	– –	365	355	**357.80**
19	– –	0.14	0.13	**0.13**	– –	31	27	**29.20**
20	– –	0.26	0.24	**0.25**	– –	54	51	**52**

more difficult than SAT instances.

The satisfiability-based algorithm is deterministic. As stated above, the CCA size affects the algorithm's execution time and, if timeout occurs, the resulting CLA size. In contrast, the heuristic algorithm is inherently nondeterministic; it generates different CLAs for different runs. The algorithm decreases the array size by repeatedly removing a test case selected at random from the current array. A test case can be removed only if the array remains a CLA after its removal; thus, which test case is removed depends strongly on earlier selections. Hence, different orders in which test cases are deleted lead to different CLAs.

7.2.3 CDA: Experimental Results

These experimental results are summarized in Table 7.3. The leftmost column shows the benchmark IDs. The rest of the table is divided into two parts representing the results of generation time and of sizes of the generated CDAs. Both parts have two sections describing the experimental results of the two proposed algorithms. For each problem instance, the average value is reported for the satisfiability-based algorithm as it is deterministic, while the maximum, minimum, and average values are reported for the heuristic algorithm.

The numbers with an asterisk (*) in the satisfiability-based algorithm's columns show that the generation did not terminate within the time limit. Because the algorithm repeatedly generates CDAs with sizes varying until the minimum one is found, CDAs that are not optimal are constructed during the course of execution. The values with an asterisk correspond to the smallest (not necessarily optimal) CDAs that were obtained within the time limit. For example, for benchmark No. 1, the algorithm took 1,557.89s to generate a CDA of size 25. However, when it was trying to generate a CDA of size 24, the algorithm exceeded the 1800s time limit. There are also some benchmark instances where the

Table 7.3: CDA: Experimental results of generating CDAs

ID	Time (second)				Size			
	SMT	Two-step			SMT	Two-step		
	value	max.	min.	avg.	value	max.	min.	avg.
1	1,557.89*	0.15	0.10	**0.13**	**25***	37	36	36.40
2	– –	0.12	0.09	**0.11**	– –	42	42	**42**
3	– –	0.21	0.15	**0.17**	– –	50	47	**48.60**
4	0.364	0.10	0.07	**0.08**	**8**	8	8	**8**
5	– –	0.17	0.12	**0.14**	– –	95	92	**94**
6	– –	0.20	0.16	**0.18**	– –	60	57	**58.20**
7	– –	3.14	2.79	**2.95**	– –	179	173	**176**
8	– –	19.87	18.73	**19.23**	– –	234	220	**227.60**
9	– –	145.14	130.48	**134.31**	– –	1,997	1,959	**1,971.40**
10	– –	1.56	1.48	**1.52**	– –	405	394	**399.40**
11	– –	0.67	0.59	**0.61**	– –	114	111	**112**
12	– –	3.16	2.94	**3.03**	– –	122	118	**120.20**
13	– –	4.82	4.76	**4.79**	– –	430	413	**422.40**
14	2.53*	0.12	0.10	**0.10**	**25***	25	25	**25**
15	– –	0.08	0.07	**0.07**	– –	51	45	**47.60**
16	– –	0.64	0.58	**0.61**	– –	189	185	**187.60**
17	– –	16.28	15.61	**15.88**	– –	517	500	**506.20**
18	– –	130.25	120.02	**124.86**	– –	860	843	**851.60**
19	– –	0.12	0.09	**0.11**	– –	53	49	**51.80**
20	– –	0.19	0.14	**0.16**	– –	102	98	**99.8**

algorithm did not find even one CDA within the time limit. The symbol "– –" is used to indicate such a case. To compare the average consumed time of the two algorithms, the better results (i.e., shorter time) are denoted in bold font. The smaller average sizes of generated CDAs are also denoted in bold font.

The satisfiability-based algorithm completed the generation for only one instance (No. 4). The algorithm was able to find small CDAs for some remaining instances (though it timed out), such as Nos. 1 and 14, whereas it failed to find even a single CDA for others. Meanwhile, the heuristic algorithm successfully generated CDAs for all benchmark instances. The execution time of the satisfiability-based algorithm was always much longer compared with that of the heuristic generation algorithm, which can be observed from the results of benchmark No. 1. It takes almost four orders of magnitude more time than the heuristic algorithm. There are two reasons for this, and they are the same as those for CLAs. One is that the algorithm continuously tries to generate CDA instances before it generates an optimal CDA. The other reason is that the satisfiability check consumes much time, especially when the checker computes the generation of a CDA with minimum size minus one.

7.2.4 Answer to RQ 1

The satisfiability-based algorithms are deterministic algorithms that can always generate optimal arrays for given SUTs. However, because of the long-time satisfiability checking, the satisfiability-based algorithms are not practical for real-world testing. Meanwhile, the heuristic algorithms have balanced capabilities with respect to running time and array sizes. However, the satisfiability-based algorithms are still an available option. Previous research [68] has shown that the execution of a test case usually costs several orders of magnitude more time than the generation of a test case for recent software systems, such as highly configurable systems. Although the satisfiability-based algorithms may take

hours to generate minimum CLAs or CDAs, using the minimum CLAs and CDAs as test suites still reduces the total time of testing for these software systems.

7.3 Experiments on Generating Arrays with $t \geq 2$

7.3.1 Experimental Setting

The results of the previous section showed that the proposed heuristic algorithms can scale to large problems when the strength t of CLAs and CDAs is two. Next, the dissertation examines the scalability of the heuristic algorithms with respect to strength of arrays.

In this experiment, the proposed heuristic algorithms were applied to the 20 benchmark instances to generate $(1, t)$-CLAs and $(1, t)$-CDAs with strength $t = 3$ and $t = 4$, respectively. As in the previous experiments, two heuristic algorithms were run 5 times for each problem.

7.3.2 CLA: Experimental Results

Table 7.4 summarizes the results of this experiment, including those obtained for $t = 2$ in Experiment 1. As in Table 7.1, the columns marked with "$|\mathcal{VI}_t|$" and "$|\mathcal{T}_a \sim \mathcal{T}_b|$" show the numbers of valid interactions and pairs of indistinguishable valid interactions of strength t, respectively. The columns labeled with "average" show the running time of the proposed algorithm and size of obtained CLAs averaged over 5 runs.

From Table 7.4, it can be observed that the sizes of generated CLAs and the generation time increased exponentially as the strength increased. For all benchmarks, the speed of growth in size was much slower than that in generation time. Another observation is that the growth in size did not change much for benchmark Nos. 4 and 14. This is because these two benchmarks have relatively small testing space, $\mathcal{R}$, and when the algorithm is

Table 7.4: CLA: Experimental results of generating CLAs with strength $2 \leq t \leq 4$

No.	t	$\lvert\mathcal{VI}_t\rvert$	$\lvert\mathcal{T}_a \sim \mathcal{T}_b\rvert$	average		No.	t	$\lvert\mathcal{VI}_t\rvert$	$\lvert\mathcal{T}_a \sim \mathcal{T}_b\rvert$	average	
				time	size					time	size
	2	102	0	0.11	26.80		2	1,058	6	0.92	59.20
1	3	324	0	0.13	80.20	11	3	14,229	231	166.15	280.40
	4	513	104	0.16	176.40		4	130,724	– –	– –	– –
	2	473	0	0.19	28		2	2,525	1,562	3.40	67.40
2	3	4,290	0	4.32	80.60	12	3	53,228	67,926	1,404.34	333.40
	4	26,728	0	178.48	199.40		4	781,771	– –	– –	– –
	2	285	69	0.17	34.40		2	1,819	93	7.41	198.40
3	3	1,650	1,221	0.96	79.80	13	3	30,031	4,313	– –	– –
	4	5,978	9,338	6.58	147.20		4	317,228	– –	– –	– –
	2	36	16	0.07	7		2	53	11	0.11	22
4	3	55	90	0.07	8	14	3	71	43	0.08	25
	4	35	46	0.07	8		4	25	0	0.08	25
	2	361	5	0.21	47.60		2	126	0	0.09	35.60
5	3	2,535	118	3.62	190	15	3	432	0	0.15	116.20
	4	11,102	1,151	48.14	565		4	729	0	0.25	305.40
	2	466	0	0.24	34.40		2	1,020	57	1.15	88.60
6	3	4,076	6	8.45	128.40	16	3	11,840	1,212	121.43	395.40
	4	23,792	183	319.47	414.60		4	89,632	13,982	– –	– –
	2	3,092	477	8.20	91		2	3,491	0	29.72	218
7	3	74,274	18,460	– –	– –	17	3	86,153	0	– –	– –
	4	1,264,002	– –	– –	– –		4	1,369,701	– –	– –	– –
	2	5,707	288	44.32	104.20		2	5,342	20	112.48	357.80
8	3	191,398	– –	– –	– –	18	3	157,949	– –	– –	– –
	4	– –	– –	– –	– –		4	– –	– –	– –	– –
	2	4,573	0	124.39	798.40		2	310	130	0.13	29.20
9	3	– –	– –	– –	– –	19	3	1,982	1,591	0.76	88.20
	4	– –	– –	– –	– –		4	7,770	10,227	5.82	216
	2	1,220	0	2.58	206.40		2	440	23	0.25	52
10	3	15,370	1	347.78	1,661.80	20	3	3,431	225	6.18	208
	4	116,350	– –	– –	– –		4	16,841	1,246	120.46	685.60

generating $(1, 3)$-CLAs for the two benchmarks, all valid test cases are already included. Thus, the changes in array sizes as well as the generation time did not vary much. With the 1800s timeout, there were six benchmark problems for which the algorithm ran out of time while generating $(1, 3)$-CLAs. The proposed algorithm also failed to generate $(1, 4)$-CLAs for ten benchmarks.

7.3.3 CDA: Experimental Results

Table 7.5 lists the results of these experiments. As shown in the table, the column marked with "$|\mathcal{VI}_t|$" shows the numbers of valid interactions of strength t; the column marked with "$|\mathcal{T} \sim T|$" shows the numbers of pairs of an interaction and interaction set that have masking relations. The columns labeled with "average" show the running time of the heuristic algorithm and sizes of obtained CDAs, respectively.

It can be observed from the values in the table that the generation time and sizes of obtained CDAs increased exponentially in general. There are also some exceptions to this observation, e.g., the benchmarks Nos. 4 and 14. The values of the two benchmarks did not change as the strength varied. This is because the CDAs include all valid test cases when $t = 2$. Both of the benchmarks have only a few parameters with small domains, and thus the testing space is relatively small. In addition, the constraints in the two benchmarks strongly restrict the testing space, so $\mathcal{R}$ equals a $(1, 2)$-CDA.

The heuristic generation algorithm failed to generate $(1, 3)$-CDAs for 7 benchmarks, and it failed to generate $(1, 4)$-CDAs for 10 benchmarks.

7.3.4 Answer to RQ 2

The heuristic algorithms for both CLAs and CDAs performed well on generating arrays with higher strengths of $t = 3$ and $t = 4$. Both algorithms generated high-strength arrays for many benchmarks even though the time limit was relatively short, i.e., 1800s.

Table 7.5: CDA: Experimental results of generating CDAs with strength $2 \leq t \leq 4$

No.	t	$\lvert\mathcal{VI}_t\rvert$	$\lvert\mathcal{T} \succ T\rvert$	average		No.	t	$\lvert\mathcal{VI}_t\rvert$	$\lvert\mathcal{T} \succ T\rvert$	average	
				time	size					time	size
	2	102	0	0.11	35		2	1,058	1,510	0.61	113.60
1	3	324	0	0.12	88.80	11	3	14,229	47,202	120.61	490.20
	4	513	832	0.12	190.20		4	130,725	– –	– –	– –
	2	473	208	0.09	41.80		2	2,525	35,156	611.02	122
2	3	4,290	3,744	0.99	99	12	3	53,228	2,370,351	– –	– –
	4	26,728	36,608	47.37	235.60		4	781,772	– –	– –	– –
	2	285	2,177	0.16	47.80		2	1,819	1,088	4.82	427.80
3	3	1,650	36,399	0.58	94	13	3	30,031	63,473	– –	– –
	4	5,978	269,705	4.23	162.60		4	– –	– –	– –	– –
	2	36	130	0.07	8		2	53	112	0.11	25
4	3	55	342	0.07	8	14	3	71	165	0.10	25
	4	35	120	0.07	8		4	25	0	0.10	25
	2	361	512	0.12	94.20		2	126	0	0.07	47
5	3	2,535	8,655	1.21	327.40	15	3	432	0	0.10	137
	4	11,102	71,160	17.75	804.40		4	729	0	0.13	368.60
	2	466	124	0.14	59		2	1,020	3,400	0.60	187.80
6	3	4,076	2,688	3.75	211.20	16	3	11,840	89,072	61.24	730.60
	4	23,792	32,948	146.08	647		4	89,623	1,186,446	– –	– –
	2	3,092	8,700	2.87	176.20		2	3,491	0	15.65	508
7	3	74,274	622,932	– –	– –	17	3	86,153	0	– –	– –
	4	1,264,002	– –	– –	– –		4	– –	– –	– –	– –
	2	5,707	3,359	17.90	224		2	5,342	10,095	119.97	849.60
8	3	191,398	– –	– –	– –	18	3	– –	– –	– –	– –
	4	– –	– –	– –	– –		4	– –	– –	– –	– –
	2	4,573	0	132.79	1,962		2	310	825	0.09	52.20
9	3	– –	– –	– –	– –	19	3	1,982	15,354	0.54	147.40
	4	– –	– –	– –	– –		4	7,770	134,882	4.36	305
	2	1,228	189	1.53	396.80		2	440	151	0.14	102
10	3	15,370	5,514	185.06	3,113.80	20	3	3,431	2,505	1.93	374.80
	4	– –	– –	– –	– –		4	16,841	20,796	42.61	1,102.80

The generation time grew exponentially with increased strength, while the array sizes increased much more gradually.

7.4 Identifying Faulty Interactions in Real-World Systems

The third experiment examined CLAs and CDAs with respect to the capability of identifying faulty interactions induced by real software bugs. By definition, CLAs and CDAs ensure that faulty interactions can be located if underlying assumptions hold. However, these assumptions may not necessarily hold in reality. The aim of this experiment was to answer the following research question:

RQ3 Can CLAs and CDAs be used to detect faulty interactions caused by actual bugs, especially when the assumptions about the number and strength of faulty interactions do not hold?

The procedure of the experiment was as follows:

Step 1 Construct SUT models for applications under test.

Step 2 Seed bugs into the source code of the application programs to create a collection of faulty versions.

Step 3 Use exhaustive testing to identify faulty interactions that are caused by the seeded bugs. The identified faulty interactions are used as correct answers.

Step 4 Use CLAs and CDAs to select test cases and locate (or estimate) faulty interactions using these test cases.

Step 5 Compare the results obtained from CLAs and CDAs with the correct answers.

In the experiments, the parameters of CLAs and CDAs were set to $d = 1$ and $t = 2$.

7.4.1 Experimental Setting

Open source software *Flex* [1] and *Gzip* [2] were chosen as applications under test, and their source code was obtained from *Software-artifact Infrastructure Repository (SIR)* [16] at the University of Nebraska–Lincoln. At SIR, each of the programs is associated with a test specification file written in the *Extended Test Specification Language* [51]. Test specification files describe all the options and patterns of the inputs to be tested, together with the requirements and specifications among the options. SIR also provides the whole testing environment for these programs, which encompasses a bug seeding facility, verified input-output sets, and a tool chain with an automatic test script generation tool.

Petke et. al. analyzed the test specification files of the two applications and provided the SUT models with constraints [52]. Their SUT models were used as the SUT models in this experiment. A summary of the two SUT models is given in Table 7.6.

The bug seeding facility provided by SIR was applied to seed bugs into *Flex* and *Gzip* in this experiment. Exactly one bug was seeded in a single version of each program.

Exhaustive testing was conducted for each application as follows. First, a CCA whose strength is equal to the total number of factors for the SUT model of the application was constructed. This CCA represents the exhaustive test suite because it consists of all valid test cases. Then, a test script was created from the CCA and applied to faulty versions of the application.

From the test outcome, faulty interactions were identified as follows. The experiment computed a *minimal* set of interactions such that 1) every interaction in the set occurs in some of the failed test cases but not in any of the passed test cases and 2) every failed test case contains at least one interaction in the set. Here, the set is regarded as minimal if no smaller set satisfies these conditions. In general, there can be more than one minimal sets, but a unique set of interactions was identified for every faulty version in our case.

As the SUT model does not completely cover the possible test space of the application,

Table 7.6: SUT models

ID	SUT	$\|\mathcal{F}\|$	$\|\phi\|$	Instructure
1	Flex	9	12	Command line options with "On" and "Off" are set as parameters with two values Command line options with different functions are set as parameters with multiple values
2	Gzip	14	61	Command line options with string input are discretized using regular expressions Command line options with file input are classified using file identifiers

Table 7.7: Seeded bugs

SUT	Name	Description
Flex	F_AA_2	array: "array[index]" to "array[index - 1]"
	F_AA_3	if condition: "var1 == var2" to "var1 = var2"
	F_AA_6	if condition: "(var1 || var2) && var3" to "var1 || (var2 && var3)"
Gzip	FAULTY_F_KL_6	value assignment: "var1 += var2" to "var1 = var2"
	FAULTY_F_KP_11	loop condition: "– –var" to "var – –"

no test case failed for some of the faulty versions. The bugs that manifested themselves are summarized in Table 7.7. The names of the bugs were given by SIR.

Then, the proposed algorithms were executed to generate a $(1, 2)$-CLA and $(1, 2)$-CDA for the SUT model. The test scripts were constructed from the CLAs and CDAs and applied to the set of faulty programs. The faulty interactions located using the CLA-based test cases were compared with the results of the exhaustive testing.

7.4.2 Experimental Results

The results of the experiments are summarized in Tables 7.8 and 7.9. The two leftmost columns show the applications and names of the bugs. The remainder of each table is divided into two parts: the exhaustive testing part and CLA (or CDA) part. Each part consists of three columns. The "#Tests" column shows the total number of test cases. The

Table 7.8: CLA: Experimental results for locating faulty interactions

SUT	Name	Exhaustive Testing			CLA		
		#Tests	#Failed	Located	#Tests	#Failed	Located
Flex	F_AA_2	500	90	1-way: {(FastSwithT, FST)} 1-way: {(FastSwithT, AlterFast)}	32	11	– –
	F_AA_3	500	468	1-way: {(Compability, off)}	32	26	1-way: {(Compability, off)}
	F_AA_6	500	20	4-way: {(Bp, Off), (FastS, FS), (Align, Off), (EqClass, Off)} 1-way: {(FastS, FullS)}	32	5	1-way: {(FastS, FullS)}
Gzip	FAULTY_F_KL_6	159	4	3-way: {(SetV, On), (Set4, On), (FileType, ASCII)}	36	2	2-way: {(Set4, On), (FileType, ASCII)}
	FAULTY_F_KP_11	159	79	1-way: {(FileType, ASCII)}	36	20	1-way: {(FileType, ASCII)}

Note:

FastSwithT = Fast Scanner with Table, FST = Fast Scanner Table, AlterFast = Alternate Fast;

Compability = Compability with AT&T Lex; Bp = Bypass use; EqClass = Equivalance Classes;

FastS = Fast Scanner, FS = Fast Scan, FullS = Full Scan;

Set_V = Set V Option; Set_4 = Set 4 Option;

Table 7.9: CDA: Experimental results for locating faulty interactions

SUT	Name	Exhaustive Testing			CDA		
		#Tests	#Failed	Located	#Tests	#Failed	Located
Flex	F_AA_2	500	90	1-way: {(FastSwithT, FST)} 1-way: {(FastSwithT, AlterFast)}	62	20	– –
	F_AA_3	500	468	1-way: {(Compability, off)}	62	47	1-way: {(Compability, off)}
	F_AA_6	500	20	4-way: {(Bp, Off), (FastS, FS), (Align, Off), (EqClass, Off)} 1-way: {(FastS, FullS)}	62	10	1-way: {(FastS, FullS)}
Gzip	FAULTY_F_KL_6	159	4	3-way: {(SetV, On), (Set4, On), (FileType, ASCII)}	47	1	3-way: {(SetF, On), (Set4, On), (FileType, ASCII)} or 3-way: {(SetV, On), (Set4, On), (FileType, ASCII)}
	FAULTY_F_KP_11	159	79	1-way: {(FileType, ASCII)}	47	27	1-way: {(FileType, ASCII)}

Note:

FastSwithT = Fast Scanner with Table, FST = Fast Scanner Table, AlterFast = Alternate Fast;

Compability = Compability with AT&T Lex; Bp = Bypass use; EqClass = Equivalance Classes;

FastS = Fast Scanner, FS = Fast Scan, FullS = Full Scan;

Set_V = Set V Option; Set_4 = Set 4 Option;

"#Failed" column shows the number of test cases that failed. The "#Located" column shows located faulty interactions. Note that the faulty interactions located by exhaustive testing are the correct answers.

For the two faulty versions denoted by F_AA_3 and FAULTY_F_KP_11, there was exactly one faulty interaction, and its strength was one. The test cases derived from the $(1, 2)$-CLAs and $(1, 2)$-CDAs successfully identified the faulty interaction.

For F_AA_2, there were two faulty interactions, both of strength one, namely, $\{$(*FastSwithT*, *FST*)$\}$ and $\{$(*FastSwithT*, *AlterFast*)$\}$. Both the CLA-based and CDA-based test cases failed to locate either of these faulty interactions. Within the CLA-based and CDA-based test cases, there was no single interaction that appeared in the failed test cases but not in the remaining 21 passed test cases. However, if it were assumed that there exist two faulty interactions of strength ≤ 2, the faulty interactions could be identified because no other interaction pairs coincide with the test outcome. This suggests that even if faulty interactions cannot be exactly located, the test outcome obtained from CLAs and CDAs may provide informative clues about them.

Similarly to F_AA_2, the case F_AA_6 also contained two faulty interactions; however, in this case, one of the faulty interactions was of strength four. In this case, both the CLA-based and CDA-based test cases correctly located one faulty interaction that of strength one. The other faulty interaction, namely, $\{$(*Bp*, *Off*), (*FastS*, *FS*), (*Align*, *On*), (*EqClass*, *Off*)$\}$, did not occur in any of the test cases because of its high strength. As a result, the four-way faulty interaction did not affect the identification of the other faulty interaction.

The case FAULTY_F_KL_6 contained one faulty interaction whose strength is three. This means that by definition, the $(1, 2)$-CLA-based test cases were not able to locate this interaction. In fact, based on the test outcome, two-way interaction $\{$(*Set4, On*), (*FileType, ASCII*)$\}$ was identified as a faulty interaction. This result was not exactly correct, but it is

useful for fault localization as it is a subset of the correct faulty interaction {(*SetV, On*), (*Set4, On*), (*FileType, ASCII*)}.

Similarly, the $(1, 2)$-CDA-based test cases were not able to locate this interaction. The results of the test cases indicated that there are no faulty 2-way interactions. The results also indicated that there were two 3-way interactions suspected to be faulty. In contrast to the CLAs, it was observed from the CDAs that no valid 2-way interactions were faulty because all valid 2-way interactions appeared in some passed test cases. By definition of CDAs, all valid interactions of strength $\leq t$ will appear in some test cases. Thus, it can be concluded that there were some faulty interactions that have strengths larger than t ($t = 2$ in this case). After checking the valid 3-way interactions, it was concluded that two 3-way interactions were suspicious as faulty. Indeed, one of the faulty interaction candidates was the truly faulty interaction.

If the number of faulty interactions and their strengths were known before testing, all faulty interactions could theoretically be identified by CLAs or CDAs. To demonstrate this, 6-CCAs were also applied to the faulty versions of the two applications. The reason for using 6-CCAs instead of $(2, 4)$-CLAs or $(2, 4)$-CDAs is that the generation of these did not terminate within 3h. In contrast, the off-the-shelf CCA generator successfully generated 6-CCAs within 30min for both SUT models. From Theorem 11 and Corollary 1, 6-CCAs are $(2, 4)$-CLAs and $(2, 4)$-CDAs. The sizes of the 6-CCAs are shown in Table 7.10.

All faulty interactions in the faulty versions of the applications Flex and Gzip were identified using the 6-CCA test suites. The number of test cases in the 6-CCA for Flex was 366, while the 6-CCA for Gzip contained 144 test cases. When compared with the exhaustive testing, the number of test cases was reduced by 26.8% and 9.4%, respectively. Note that within the two 6-CCAs, there were a number of redundant test cases for fault identification. If minimum $(2, 4)$-CLAs and $(2, 4)$-CDAs had been used, the number of

Table 7.10: $(2, 4)$-CLA and $(2, 4)$-CDA (6-CCA): Experimental results for locating faulty interactions

SUT	Name	Faulty Interactions	# Test			
			Exhaustive Testing	$(1, 2)$-CLA	$(1, 2)$-CDA	6-CCA used as $(2, 4)$-CLA and CDA
Flex	F_AA_2	1-way: {(FastSwithT, FST)} 1-way: {(FastSwithT, AlterFast)}	500	32	62	366
	F_AA_3	1-way: {(Compability, off)}	500	32	62	366
	F_AA_6	4-way: {(Bp, Off), (FastS, FS), (Align, Off), (EqClass, Off)} 1-way: {(FastS, FullS)}	500	32	62	366
Gzip	FAULTY_F_KL_6	3-way: {(SetV, On), (Set4, On), (FileType, ASCII)}	159	36	47	144
	FAULTY_F_KP_11	1-way: {(FileType, ASCII)}	159	36	47	144

test cases could have been further reduced.

7.4.3 Answer to RQ 3

The test cases derived from CLAs and CDAs may fail to locate faulty interactions if there are more than d faulty interactions or faulty interactions have strength greater than t; however, even in such cases, they can still provide information useful for localization of faulty interactions.

CHAPTER 8

RELATED WORK

CIT has been widely used for many years. There have been many reports on the usage of CIT in real-world testing. Early reports include that of Tatsumi, Fujitsu Ltd in [60]. IBM Global Business Services reported their experience using CIT as solutions for testing two insurance sector clients in North America [27]. The researchers from IBM Haifa also shared their stories [26] about using CIT for a customer in the telecommunication industry and for IBM internal. Microsoft Corporation has been using their CIT tool PICT [14] since 2000. Am empirical study on using CIT in real-world systems was published in [25].

In CIT, the most studied test suites are in the forms of CAs and CCAs. Many approaches have been proposed to generate CAs and CCAs. AETG [8] was an early method for generating CAs. The AETG algorithm is a greedy algorithm that adopts the strategy of one-test-at-a-time to generate small CAs. In contrast, the greedy algorithm IPOG [38] adopts the one-parameter-at-a-time strategy. The IPOG algorithm is used more frequently as it can generate near-minimum CAs in a very short time. There also exist several meta-heuristic algorithms for generating CAs and CCAs, such as simulated annealing algorithms [19, 20] and tabu search [18]. The satisfiability-based constructions of CAs and CCAs can be found in [23, 45, 67]. Other research, such as [62–64], explored system

modeling techniques and test scheduling techniques for CIT. [6, 7, 21, 36, 42] combined CIT with the field of artificial intelligence.

LAs and DAs were first introduced by Colbourn and McClary in [10]. They analyzed the mathematical properties of these arrays. As in [10], most studies on LAs and DAs focus on their mathematical aspects [12, 13, 41, 54, 57, 58]. The application to screening experiments for TCP throughput in a mobile wireless network was reported in [3, 11]. Other types of arrays that are intended for fault location include *error locating arrays* [43, 44] and *consecutive detecting arrays* [56].

The concept of CLAs was first introduced in [28]. Later, a computational construction algorithm was proposed in [31]. In [32], the results of applying CLAs to fault identification for real-world programs were reported.

In [29], (d, t)-CDAs were introduced for the first time, together with a construction algorithm using an SMT-solver. The two-step heuristic algorithm was first proposed in [33]. [30] extended two earlier works: [29] and [33]. In the paper, the implementation of the algorithm introduced in [33] was improved, and a new set of experiments were conducted to compare the two different algorithms with the new implementations.

This dissertation expands the two papers [30,32] with examples and new experimental results.

There are many other approaches to faulty interaction localization without using CLAs, CDAs, or other related arrays. One of such approach is the use of adaptive testing [4, 5, 39, 48–50, 65, 69]. In adaptive testing, when a failure is encountered, new test cases are adaptively generated and executed to narrow down possible causes. Meanwhile, testing using CLAs and CDAs is nonadaptive in the sense that test outcomes do not alter future test plans. A clear benefit of using nonadaptive testing is that the execution of test suites, which is often the most time-consuming part of the whole testing process, can be parallelized.

CLAs, CDAs, and other arrays of similar kinds are intended to provide sufficient test outcomes to uniquely identify faulty interactions, while some studies have attempted to infer faulty interactions from insufficient information with, for example, machine learning. The studies along this line include [47, 55, 68].

Other approaches to identification of faulty interactions can be found in a recent survey [17].

CHAPTER 9

CONCLUSION

9.1 Conclusion

This dissertation introduced CLAs and CDAs, which incorporate constraints among test parameters into LAs and DAs, respectively. CLAs and CDAs generalize LAs and DAs so that localization of faulty interactions can be performed for systems with constraints. Several properties of CLAs and CDAs were proved as well as those that relate CLAs and CDAs with each other and with other array structures, such as CCAs. Then, this dissertation proposed two generation algorithms. The first algorithm generates optimal CLAs and CDAs using an off-the-shelf satisfiability solver. The second algorithm is heuristic and generates near-optimal CLAs and CDAs in a reasonable time. The experimental results of the satisfiability-based algorithm showed that it can generate CLAs and CDAs with minimum sizes as long as sufficient generation time is allowed. The results also showed that the heuristic algorithm scales to problems of practical sizes. The final experiment showed that both CLAs and CDAs can be used to identify faulty interactions in real-world testing if the number of faulty interactions and their strengths do not exceed the limitation of d and t for the arrays.

9.2 Future Work

There are several possible directions for future work. One is to improve the two algorithms proposed in this dissertation. In the dissertation, the satisfiability-based algorithms adopted the *naïve matrix model* as a representation of a test suite. However, using other encoding techniques, such as symmetry breaking, might reduce the solution space and thus shorten the generation time of the algorithms. In the heuristic algorithms, the test cases to be checked are chosen at random. Different orders of choosing test cases lead to different sizes of resulting arrays and different generation times. Finding the best order of choosing test cases can improve the algorithms as well. The development of new algorithms for CLA and CDA construction also deserves further studies. Both meta-heuristic search and greedy heuristics may be promising because they have proved to be useful for the construction of CCAs. Another direction is comparing the capabilities of fault identification between adaptive testing methods with non-adaptive testing methods with respect to accuracy of fault identification, testing cost, etc.

BIBLIOGRAPHY

[1] The Fast Lexical Analyzer - scanner generator for lexing in C and C++, https://github.com/westes/flex.

[2] GNU Gzip, https://www.gnu.org/software/gzip/.

[3] A. N. Aldaco, C. J. Colbourn, and V. R. Syrotiuk. Locating arrays: A new experimental design for screening complex engineered systems. *SIGOPS Oper. Syst. Rev.*, 49(1):31–40, Jan. 2015.

[4] P. Arcaini, A. Gargantini, and M. Radavelli. Efficient and guaranteed detection of t-way failure-inducing combinations. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 200–209, 2019.

[5] J. Bonn, K. Foegen, and H. Lichter. A framework for automated combinatorial test generation, execution, and fault characterization. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–233, 2019.

[6] J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. A combinatorial approach to explaining image classifiers. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page preprint, 2021.

[7] J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. A combinatorial approach to testing deep neural network-based autonomous driving systems. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page preprint, 2021.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. on Software Engineering*, 23(7):437–444, July 1997.

[9] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, 58:121–167, 2004.

[10] C. J. Colbourn and D. W. McClary. Locating and detecting arrays for interaction faults. *Journal of Combinatorial Optimization*, 15(1):17–48, Jan. 2008.

[11] C. J. Colbourn and V. R. Syrotiuk. Coverage, location, detection, and measurement. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 19–25, April 2016.

[12] C. J. Colbourn and V. R. Syrotiuk. On a combinatorial framework for fault characterization. *Mathematics in Computer Science*, 12(4):429–451, Dec 2018.

[13] R. Compton, M. T. Mehari, C. J. Colbourn, E. De Poorter, and V. R. Syrotiuk. Screening interacting factors in a wireless network testbed using locating arrays. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 650–655, April 2016.

[14] J. Czerwonka. Pairwise testing in real world. practical extensions to test case generators. In *Proc. of the 24th Annual Pacific Northwest Software Quality Conference*, pages 419–430, Oct. 2006.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[17] T. Friedrichs, K. Fögen, and H. Lichter. A comparison infrastructure for fault characterization algorithms. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 201–210, 2020.

[18] P. Galinier, S. Kpodjedo, and G. Antoniol. A penalty-based tabu search for constrained covering arrays. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 1288–1294, New York, NY, USA, 2017. Association for Computing Machinery.

[19] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. *In International Symposium on Search Based Software Engineering (SSBSE)*, pages 13–22, 2009.

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

[21] C. Gladisch, C. Heinzemann, M. Herrmann, and M. Woehrle. Leveraging combinatorial testing for safety-critical computer vision datasets. 06 2020.

[22] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification and Reliability*, 15(3):167–199, Sept. 2005.

[23] B. Hnich, S. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem. pages 172–186, 06 2004.

[24] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2):199–219, 2006.

[25] L. Hu, W. E. Wong, D. R. Kuhn, and R. N. Kacker. How does combinatorial testing perform in the real world: an empirical study. *Empirical Software Engineering*, 25(4):2661–2693, 2020.

[26] IBM Haifa, https://research.ibm.com/haifa/dept/_svt/papers/CTD_Introduction.pdf.

[27] IBM Global Business Services, https://www.ibm.com/downloads/cas/GANDBVKQ.

[28] H. Jin, T. Kitamura, E.-H. Choi, and T. Tsuchiya. A satisfiability-based approach to generation of constrained locating arrays. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 285–294, April 2018.

[29] H. Jin, C. Shi, and T. Tsuchiya. Constrained detecting arrays for fault localization in combinatorial testing. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1971–1978, New York, NY, USA, 2020. Association for Computing Machinery.

[30] H. Jin, C. Shi, and T. Tsuchiya. Constrained detecting arrays: Mathematical structures for fault identification in combinatorial interaction testing, 2021.

[31] H. Jin and T. Tsuchiya. Deriving fault locating test cases from constrained covering arrays. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 233–240, Dec 2018.

[32] H. Jin and T. Tsuchiya. Constrained locating arrays for combinatorial interaction testing. *Journal of Systems and Software*, 170:110771, 2020.

[33] H. Jin and T. Tsuchiya. A two-step heuristic algorithm for generating constrained detecting arrays for combinatorial interaction testing. In *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 219–224, 2020.

[34] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, pages 91–95, 2002.

[35] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[36] D. R. Kuhn, R. N. Kacker, Y. Lei, and D. E. Simos. Combinatorial methods for explainable ai. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 167–170, 2020.

[37] D. R. Kuhn and D. R. Wallace. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[38] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[39] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, pages 102–105, 2012.

[40] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *Proc. of the 30th International Conference on Automated Software Engineering (ASE)*, pages 494–505. ACM/IEEE, 2015.

[41] X.-N. Lu and M. Jimbo. Arrays for combinatorial interaction testing: a review on constructive approaches. *Japanese Journal of Statistics and Data Science*, 2:641–667, 12 2019.

[42] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 614–618, 2019.

[43] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2010.

[44] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using elas, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2010.

[45] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 95-A(9):1501–1505, 2012.

[46] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43:11:1–11:29, feb 2011.

[47] K. Nishiura, E. Choi, and O. Mizuno. Improving faulty interaction localization using logistic regression. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 138–149, July 2017.

[48] X. Niu, C. Nie, Y. Lei, and A. T. S. Chan. Identifying failure-inducing combinations using tuple relationship. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 271–280, March 2013.

[49] X. Niu, C. Nie, H. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang. An interleaving approach to combinatorial testing and failure-inducing interaction identification. *IEEE Trans. Softw. Eng.*, 46(6):584–615, June 2020.

[50] X. Niu, H. Wu, N. Changhai, Y. Lei, and X. Wang. A theory of pending schemas in combinatorial testing. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[51] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31:676–686, 06 1988.

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

[53] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proc. of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–264. ACM, 2011.

[54] S. A. Seidel, K. Sarkar, C. J. Colbourn, and V. R. Syrotiuk. Separating interaction effects using locating and detecting arrays. In C. Iliopoulos, H. W. Leong, and W.-K. Sung, editors, *Combinatorial Algorithms*, pages 349–360, Cham, 2018. Springer International Publishing.

[55] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and D. R. Kuhn. Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 620–623, 2012.

[56] C. Shi, L. Jiang, and A. Tao. Consecutive detecting arrays for interaction faults. *Graphs and Combinatorics*, 36(4):1203–1218, 2020.

[57] C. Shi, Y. Tang, and J. Yin. The equivalence between optimal detecting arrays and super-simple OAs. *Designs, Codes and Cryptography*, 62(2):131–142, Feb. 2012.

[58] C. Shi, Y. Tang, and J. Yin. Optimal locating arrays for at most two faults. *Science China Mathematics*, 55(1):197–206, Jan. 2012.

[59] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proc. of 28th Annual International Computer Software and Applications Conference (COMPSAC '04)*, pages 71–77, 2004.

[60] K. Tatsumi. Test case design support system. In *Proc. of International Conference on Quality Control (ICQC'87)*, pages 615–620, 1987.

[61] T. Tsuchiya. Using binary decision diagrams for constraint handling in combinatorial interaction testing. *CoRR*, abs/1907.01779, 2019.

[62] R. Tzoref. Comprehension and evolution of combinatorial models and test plans. *SIGSOFT Softw. Eng. Notes*, 45(3):23–24, jul 2020.

[63] R. Tzoref-Brill and S. Maoz. Syntactic and semantic differencing for combinatorial models of test designs. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 621–631. IEEE Press, 2017.

[64] R. Tzoref-Brill and S. Maoz. Modify, enhance, select: Co-evolution of combinatorial models and test plans. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 235–245, New York, NY, USA, 2018. Association for Computing Machinery.

[65] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *2010 10th International Conference on Quality Software*, pages 495–502, July 2010.

[66] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman. A survey of constrained combinatorial testing, 2019.

[67] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 614–624, 2016.

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

[69] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 331–341, 2011.