The Journal of Systems & Software 170 (2020) 110771

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Constrained locating arrays for combinatorial interaction testing

Hao Jin *, Tatsuhiro Tsuchiya

Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita-Shi, Osaka 565-0871, Japan

ARTICLE INFO

Article history:
Received 30 May 2019
Received in revised form 25 June 2020
Accepted 31 July 2020
Available online 5 August 2020

MSC:
94C12
05B30
68R05

Keywords:
Combinatorial interaction testing
Locating arrays
Covering arrays
Software testing

ABSTRACT

This paper introduces the notion of *Constrained Locating Arrays (CLAs)*, mathematical objects which can be used for fault localization in software testing. CLAs extend ordinary locating arrays to make them applicable to testing of systems that have constraints on test parameters. Such constraints are common in real-world systems; thus CLA enhances the applicability of locating arrays to practical testing problems. The paper also proposes an algorithm for constructing CLAs. Experimental results show that the proposed algorithm scales to problems of practical sizes.

1. Introduction

Combinatorial interaction testing is a well-known strategy for software testing. In the strategy, a *System Under Test (SUT)* is modeled as a finite set of test parameters or *factors* and every interaction of interest is exercised by at least one test. Empirical results suggest that testing interactions involving a fairly small number of factors, typically two or three, suffices to reveal most of latent faults. Many studies have been developed to construct small test sets for combinatorial interaction testing. Such test sets are often called *Covering Arrays (CAs)*. Surveys on these studies can be found in, for example, Colbourn (2004), Grindal et al. (2005) and Nie and Leung (2011).

An important direction of extending the capability of combinatorial interaction testing is to add fault localization capability to it. *Locating Arrays (LAs)* can be used as test suites that provide this capability (Colbourn and McClary, 2008). In Colbourn and McClary (2008) LAs of a few different types are defined. For example, a (d, t)-LA enables us to locate a set of d failure-triggering t-way interactions using the test outcome.

The purpose of this paper is to extend the notion of LAs to expand the applicability to practical testing problems. Specifically, we propose *Constrained Locating Arrays (CLAs)* which can be used to detect and locate failure-triggering interactions in the presence of *constraints*. Constraints, which prohibit some particular tests, are common in real-world systems. Constraint handling has been well studied in the field of combinatorial interaction testing (Wu et al., 2019a). The main focus of the previous studies is on constructing test sets, often called a *Constrained Covering Array (CCA)*, that consist only of constraint-satisfying tests and cover all interactions that can occur in constraint-satisfying tests. CLAs add the ability of fault localization to CCAs.

However, CLAs require additional considerations about constraints. Specifically, constraints may make it impossible to distinguish a failure-triggering interaction or set of such interactions from another; hence a special treatment must be needed to deal with such an inherently indistinguishable pair. By extending LAs with the concept of *distinguishability*, we provide the formal definition of CLAs. We also propose a generation method for CLAs and demonstrate that the generation method can scale to problems of practical sizes.

The rest of the paper is organized as follows. Section 2 describes the SUT model and the definition of locating arrays, as well as some related notions. Section 3 presents the definition of CLAs and some basic theorems about them. Section 4 presents a computational method for generating CLAs. Section 5 shows experimental results obtained by applying the method to a number of problem instances. Section 6 summarizes related work. Section 7 concludes the paper with possible future directions of work.

* Corresponding author.
E-mail address: k-kou@ist.osaka-u.ac.jp (H. Jin).

https://doi.org/10.1016/j.jss.2020.110771

factors	F_1: Display	F_2: Email Viewer	F_3: Camera	F_4: Video Camera	F_5: Video Ringtones
values	0 : 16 MC 1 : 8 MC 2 : BW	0 : Graphical 1 : Text 2 : None	0 : 2 MP 1 : 1 MP 2 : None	0 : Yes 1 : No	0 : Yes 1 : No
constraints	$F_2 = 0 \Rightarrow F_1 \neq 2$ Graphical email viewer requires color display $F_3 = 0 \Rightarrow F_1 \neq 2$ 2 Megapixel camera requires color display $F_2 = 0 \Rightarrow F_3 \neq 0$ Graphical email viewer not supported with 2 Megapixel camera $F_1 = 1 \Rightarrow F_3 \neq 0$ 8 Million color display does not support 2 Megapixel camera $F_4 = 0 \Rightarrow (F_3 \neq 2 \wedge F_1 \neq 2)$ Video camera requires camera and color display $F_5 = 0 \Rightarrow F_4 = 0$ Video ringtones cannot occur with No video camera $\neg(F_1 = 0 \wedge F_2 = 1 \wedge F_3 = 0)$ The combination of 16 Million colors, Text email viewer and 2 Megapixel camera will not be supported				

Fig. 1. Example of an SUT Cohen et al. (2008).

2. Preliminaries

2.1. SUT models, tests, and interactions

An SUT is modeled as $\langle \mathcal{F}, \mathcal{S}, \phi \rangle$ where $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$ is a set of factors, $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ is a set of domains for the factors, and $\phi : S_1 \times \cdots \times S_k \rightarrow \{true, false\}$ is a mapping that represents constraints. Each domain S_i consists of two or more consecutive integers ranging from 0; i.e., $S_i = \{0, 1, \ldots, |S_i| - 1\}$ ($|S_i| > 1$). A *test* is an element of $S_1 \times S_2 \times \cdots \times S_k$. A test $\boldsymbol{\sigma}$ is *valid* if and only if (iff) it satisfies the constraints ϕ, i.e.., $\phi(\boldsymbol{\sigma}) = true$. Given an SUT, we denote the set of all valid tests as $\mathcal{R}$. For a set of t ($0 \leq t \leq k$) factors, $\{F_{i_1}, \ldots, F_{i_t}\} \subseteq \mathcal{F}$, the set $\{(i_1, \sigma_1), \ldots, (i_t, \sigma_t)\}$ such that $\sigma_j \in S_j$ for all j ($1 \leq j \leq t$) is a *t-way interaction* or an interaction of *strength* t. Hence a test contains or *covers* $\binom{k}{t}$ t-way interactions. Note that a k-way interaction $\{(1, \sigma_1), \ldots, (k, \sigma_k)\}$ and a test $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_k)$ can be treated interchangeably. Thus we write $T \subseteq \boldsymbol{\sigma}$ iff a test $\boldsymbol{\sigma}$ covers an interaction T. It should be noted that the only 0-way is the empty set. We use $\sqcup$, instead of $\emptyset$, to denote the 0-way interaction.

Constraints may make it impossible to test some interactions. These interactions cannot be covered by any valid tests. We call such an interaction *invalid*. Formally, an interaction T is *valid* if $T \subseteq \boldsymbol{\sigma}$ for some valid test $\boldsymbol{\sigma} \in \mathcal{R}$; it is *invalid*, otherwise.

As a running example, consider a classic cell-phone example taken from Cohen et al. (2008) (Fig. 1). This SUT model has five factors which have three or two values in their domains. The constraints consist of seven parts. Test (1, 0, 1, 1, 1), for example, is valid, whereas test (1, 0, 0, 0, 1) is not valid (invalid) because it violates the third and fourth constraints. Similarly, two-way interaction $\{(1, 1), (2, 0)\}$ is valid, since it occurs in valid test (1, 0, 1, 1, 1). On the other hand, $\{(2, 0), (3, 0)\}$ is invalid, since it violates constraint $F_2 = 0 \Rightarrow F_3 \neq 0$ and thus never occurs in any valid tests.

A *test suite* is defined as a (possibly empty) collection of tests and thus can be represented as an $N \times k$ array A when the number of tests is N. For such an array A and interaction T, we let $\rho_A(T)$ denote the set of tests (rows) of A in which the interaction is covered. For a set of interactions $\mathcal{T}$, we define $\rho_A(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \rho_A(T)$. We use $\emptyset$ to denote an empty set of interactions. Clearly $\rho_A(\emptyset) = \emptyset$. (By comparison, $\rho_A(\sqcup)$ is the set of all rows of A.)

An interaction is either *faulty* or not. A *fault* is an interaction that is faulty. A *failure* is caused by a fault: the result of executing a test $\boldsymbol{\sigma}$ is *fail* iff $\boldsymbol{\sigma}$ covers at least one faulty interaction; otherwise the result is *pass*. Hence the result of executing a test suite A is a vector of size N, each element being either pass or fail.

2.2. Covering arrays and locating arrays

When there are no constraints, i.e., $\phi(\boldsymbol{\sigma}) = true$ for any test $\boldsymbol{\sigma} \in S_1 \times \cdots \times S_k$, a *Covering Array* (CA) can be used to detect the existence of fault-triggering interactions of a given strength t or less. Let $\mathcal{I}_t$ be the set of all t-way interactions. Formally, a t-CA is defined by the following condition:

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

On the other hand, a *Locating Array* (LA) can be used to locate the set of faulty interactions. Colbourn and McClary introduced a total of six types of LAs in Colbourn and McClary (2008). The definitions of the two most basic types of LAs are shown below.

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

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

The definition of other two types of LAs, namely $(d, \bar{t})$-LAs and $(\bar{d}, \bar{t})$-LAs, requires the notion of *independence* (Colbourn and McClary, 2008). Let $\overline{\mathcal{I}_t}$ be the set of all interactions of strength at most t, i.e., $\overline{\mathcal{I}_t} = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_t$. A set of interactions (interaction set) $\mathcal{T} \subseteq \overline{\mathcal{I}_t}$ is *independent* iff there do not exist two interactions $T, T' \in \mathcal{T}$ with $T \subset T'$. For example, consider a set of two interactions $\{\{(1, 1)\}, \{(1, 1), (2, 0)\}\}$ ($\subseteq \overline{\mathcal{I}_2}$) for the running example. This interaction set is not independent because $\{(1, 1)\} \subset \{(1, 1), (2, 0)\}$. Note that if two interactions T, T' are both faulty and $T \subset T'$, then the failure caused by T always masks the failure caused by T'. Because of this, it is natural to limit the scope of fault localization to independent interaction sets. Based on $\overline{\mathcal{I}_t}$ and the notion of independent interaction sets, the two types of LAs are defined as follows.

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

Fig. 2. (1,2)-LA for the running example. Constraints are not taken into account.

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

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

We do not consider the remaining two types of locating arrays, namely $(d, \hat{t})$-LAs and $(\bar{d}, \hat{t})$-LAs, because they either exist in trivial cases or otherwise are equivalent to $(d, \bar{t})$- and $(\bar{d}, \bar{t})$-LAs.

Fig. 2 shows a $(1, 2)$-LA for the running example shown in Fig. 1. Let A be the LA and $\boldsymbol{\sigma}_i$ $(1 \le i \le N = 15)$ be the ith row. If the pass/fail result were obtained for all these tests, any faulty single two-way interaction could be identified. For example, if only the first test $\boldsymbol{\sigma}_1$ failed, then the faulty interaction would be determined to be $\{(2, 0), (3, 0)\}$, because $\rho(\mathcal{T}) = \{\boldsymbol{\sigma}_1\}$ holds only for $\mathcal{T} = \{\{(2, 0), (3, 0)\}\}$, provided that $|\mathcal{T}| = 1$ and $|T| = 2$ for $T \in \mathcal{T}$. However, this array cannot be used for testing the system because of the constraints. For example, $\boldsymbol{\sigma}_1$ is not valid and thus cannot be executed in reality.

3. Constrained locating arrays

3.1. Definitions of CLAs

In the presence of constraints, a test suite must consist of only valid tests. From now on, we assume that an array A representing a test suite consists of a (possibly empty) set of valid tests. In practice, this problem has been circumvented by, instead of CAs, using Constrained Covering Arrays (CCAs). Let $\mathcal{VI}_t$ be the set of all valid t-way interactions. Then a CCA of strength t, denoted as t-CCA, is defined as follows.

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

In words, a t-CCA is an array that covers all valid interactions of strength t. It is easy to see that a t-CCA, $t \ge 1$ is a $(t-1)$-CCA. Therefore, the above definition is equivalent to:

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

Fig. 3 shows a 2-CCA for the running example.

When incorporating constraints into LA, it is crucial to take into consideration, in addition to the presence of invalid interactions, the fact that constraints may make it impossible to identify some set of faulty interactions, which could be identified if no constraints existed. This requires us the notion of *distinguishability* to formally define CLAs.

Definition 1. A pair of sets of valid interactions, $\mathcal{T}_1$ and $\mathcal{T}_2$, are distinguishable iff $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ for some array A consisting of valid tests.

For the running example, $\mathcal{T}_1 = \{\{(1, 0), (3, 0)\}\}$, $\mathcal{T}_2 = \{\{(2, 2), (3, 0)\}\}$ are not distinguishable (indistinguishable), since any valid test contains either both of the two-way interactions or none of them. That is, tests that cover exactly one of the two interaction sets (e.g., (0 1 0 0 0) or (1 2 0 0 0)) are all invalid. Hence no array A exists such that $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$.

It should be noted that even if there are no constraints, there can be some indistinguishable pairs of interaction sets. In the running example, two interaction sets $\{\{(4, 0)\}, \{(4, 1)\}\}$, $\{\{(5, 0)\}, \{(5, 1)\}\}$ are indistinguishable even if the constraints were removed, because any test has 0 or 1 on factors F_4 and F_5. Another extreme case is when $\mathcal{T}_1$ and $\mathcal{T}_2$ are identical. Clearly, identical interactions are always indistinguishable.

Definition 2. Let $d \ge 0$ and $0 \le t \le k$. 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 tests or no rows is a (d, t)-, $(\bar{d}, t)$-, $(d, \bar{t})$- or $(\bar{d}, \bar{t})$-CLA iff the corresponding condition shown below holds.

(d, t)-CLA $\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_t$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| = d$ and $\mathcal{T}_1, \mathcal{T}_2$ are distinguishable: $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

$(\bar{d}, t)$-CLA $\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_t$ such that $0 \le |\mathcal{T}_1| \le d, 0 \le |\mathcal{T}_2| \le d$ and $\mathcal{T}_1, \mathcal{T}_2$ are distinguishable: $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

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

$(\bar{d}, \bar{t})$-CLA $\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ such that $0 \le |\mathcal{T}_1| \le d, 0 \le |\mathcal{T}_2| \le d$ and $\mathcal{T}_1, \mathcal{T}_2$ are independent and distinguishable: $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$

(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 $\le d$) faulty interactions, then the test outcome obtained by executing all tests in A will be different from the one that would be obtained when the SUT had a different set of d (or $\le d$) faulty interactions, unless the two interaction sets are not distinguishable.

The algorithm to identify faulty interactions is directly obtained from the definition. Given a $(\bar{d}, \bar{t})$-CLA A, for example, the algorithm amounts to checking, for each independent $\mathcal{T} \subseteq \overline{\mathcal{VI}_t}$ such that $0 \le |\mathcal{T}| \le d$, if $\rho_A(\mathcal{T})$ is equal to the set of failed tests. When the assumptions about the number $(\le d)$ and strength $(\le t)$ of faulty interactions hold, $\mathcal{T}$ that satisfies the condition is either (1) the set of faulty interactions or (2) a set of interactions that is indistinguishable from the former.

3.2. Examples of CLAs

Here we show $(1, 1)$-, $(\bar{2}, 1)$-, $(1, \bar{2})$- and $(\bar{2}, \bar{2})$-CLAs for the running SUT example. Figs. 4, 5, 6 and 7 respectively show these CLAs. The sizes (i.e., the number of rows) of these arrays are 5, 12, 15 and 28. The number of valid tests for the running example is 31; thus these CLAs, except the $(\bar{2}, \bar{2})$-CLA, are considerably smaller than the array that consists of all valid tests. On the other hand, the $(\bar{2}, \bar{2})$-CLA is almost as large as the exhaustive one. The three missing valid tests are (0, 2, 1, 0, 1), (0, 2, 1, 1, 1) and (1, 1, 1, 1, 1).

One can verify that these are indeed CLAs by checking the necessary and sufficient conditions using the facts shown below.

$\boldsymbol{\sigma}_1$	0	0	1	0	1
$\boldsymbol{\sigma}_2$	0	0	2	1	1
$\boldsymbol{\sigma}_3$	0	1	1	0	0
$\boldsymbol{\sigma}_4$	0	2	0	0	0
$\boldsymbol{\sigma}_5$	0	2	0	1	1
$\boldsymbol{\sigma}_6$	1	0	1	0	0
$\boldsymbol{\sigma}_7$	1	1	1	0	0
$\boldsymbol{\sigma}_8$	1	1	2	1	1
$\boldsymbol{\sigma}_9$	1	2	2	1	1
$\boldsymbol{\sigma}_{10}$	2	1	2	1	1
$\boldsymbol{\sigma}_{11}$	2	2	1	1	1

Fig. 3. 2-CCA for the running example.

$\boldsymbol{\sigma}_1$	0	0	1	1	1
$\boldsymbol{\sigma}_2$	0	2	0	0	1
$\boldsymbol{\sigma}_3$	1	1	1	0	0
$\boldsymbol{\sigma}_4$	1	2	2	1	1
$\boldsymbol{\sigma}_5$	2	1	1	1	1

Fig. 4. (1, 1)-CLA for the running example.

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

Fig. 5. $(\overline{2}, 1)$-CLA for the running example.

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

Fig. 6. $(1, \overline{2})$-CLA for the running example.

For the running example, all interactions of strength ≤ 2 are valid, except ten two-way interactions listed below.

$\{(1,2),(2,0)\}$ $\{(1,1),(3,0)\}$ $\{(1,2),(3,0)\}$ $\{(1,2),(4,0)\}$
$\{(1,2),(5,0)\}$ $\{(2,0),(3,0)\}$ $\{(2,1),(3,0)\}$ $\{(3,2),(4,0)\}$
$\{(3,2),(5,0)\}$ $\{(4,1),(5,0)\}$

For the example, all pairs $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_{t=1}$ such that $\mathcal{T}_1 \neq \mathcal{T}_2$ and $|\mathcal{T}_1| = |\mathcal{T}_2| = d = 1$ are distinguishable. That is, any pair of distinct one-way interactions are distinguishable. Fig. 8 shows pairs of interaction sets that are *not* distinguishable for the other parameters $d(\overline{d})$, $t(\overline{t})$.

Using the 2-CCA in Fig. 3 and the $(1, \overline{2})$-CLA in Fig. 6 as examples, we illustrate how CLAs can be used to locate faulty interactions. Suppose that the system has exactly one faulty interaction which is of strength two. For presentation simplicity, if $\{T\}$ and $\{T'\}$ are indistinguishable from each other for interactions T and T', we say that T and T' are indistinguishable.

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

Fig. 7. $(\overline{2}, \overline{2})$-CLA for the running example.

First let us assume that the faulty interaction is $T_f = \{(2,2), (3,2)\}$. When the 2-CCA is used as a test suite, $\boldsymbol{\sigma}_9$ is the only test that fails. This test outcome is identical to when another interaction $T_f' = \{(1,1),(2,2)\}$, instead of T_f, is faulty. As a result, it is impossible to determine which one of the two interactions is faulty from the test outcome. In contrast, when the $(1, \overline{2})$-CLA is used, failing tests are different between T_f and any other two-way interaction. For example, only $\boldsymbol{\sigma}_7$ fails when T_f is faulty, whereas $\boldsymbol{\sigma}_{12}$ fails when T_f' is faulty. Hence, it can be safely concluded that the faulty interaction is T_f. By definition, $(1, \overline{2})$-CLAs ensure accurate fault detection unless there is an interaction that is different but indistinguishable from the faulty one.

Now consider $T_g = \{(2,2),(3,0)\}$ and $T_g' = \{(1,0),(3,0)\}$ which are indistinguishable from each other. When the $(1, \overline{2})$-CLA is used as a test suite, the failing tests will be $\boldsymbol{\sigma}_4$, $\boldsymbol{\sigma}_5$, and $\boldsymbol{\sigma}_6$ whenever either one of the two interactions is faulty. However, this test outcome never occurs if another interaction that is distinguishable from T_g is faulty. In general, CLAs guarantee that the test outcome is always different between a pair of interaction sets that are distinguishable. On the other hand, CCAs provide no such guarantee. For example, when the 2-CCA is used, $\boldsymbol{\sigma}_4$ and $\boldsymbol{\sigma}_5$ fail if T_g or T_g' is faulty. This test outcome also arises when $\{(1,0),(2,2)\}$ is faulty; thus it is not possible to narrow down the candidates for the fault to T_g and T_g'.

{{(1, 0)}}, {{(1, 0)}, {(3, 0)}}	{{(2, 2)}}, {{(2, 2)}, {(3, 0)}}
{{(4, 0)}}, {{(4, 0)}, {(5, 0)}}	{{(4, 1)}}, {{(1, 2)}, {(4, 1)}}
{{(4, 1)}}, {{(3, 2)}, {(4, 1)}}	{{(5, 1)}}, {{(1, 2)}, {(5, 1)}}
{{(5, 1)}}, {{(3, 2)}, {(5, 1)}}	{{(5, 1)}}, {{(4, 1)}, {(5, 1)}}
{{(1, 2)}, {(4, 1)}}, {{(3, 2)}, {(4, 1)}}	{{(1, 2)}, {(5, 1)}}, {{(3, 2)}, {(5, 1)}}
{{(1, 2)}, {(5, 1)}}, {{(4, 1)}, {(5, 1)}}	{{(3, 2)}, {(5, 1)}}, {{(4, 1)}, {(5, 1)}}
{{(4, 0)}, {(4, 1)}}, {{(4, 0)}, {(5, 1)}}	{{(4, 0)}, {(4, 1)}}, {{(5, 0)}, {(5, 1)}}
{{(4, 0)}, {(5, 1)}}, {{(5, 0)}, {(5, 1)}}	

(a) $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{VI}_{t=1}$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| \leq d = 2$ and $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinct but indistinguishable.

{{(1, 0), (3, 0)}}, {{(2, 2), (3, 0)}}	{{(1, 2), (4, 1)}}, {{(1, 2), (5, 1)}}
{{(3, 2), (4, 1)}}, {{(3, 2), (5, 1)}}	

(b) $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_{t=2}}$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| = d = 1$ and $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinct but indistinguishable. (Since $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$, $\mathcal{T}_1, \mathcal{T}_2$ are trivially independent.)

{ {(1,0)} }, { {(1,0)}, {(3,0)} }
{ {(1,2), (4,1)} }, { {(1,2), (4,1)}, {(1,2), (5,1)} }
{ {(1,0), (2,0)}, {(1,1), (2,0)} }, { {(1,2), (4,1)}, {(1,2), (5,1)} }
...

(c) Some examples of $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_{t=2}}$ such that $|\mathcal{T}_1| = |\mathcal{T}_2| \leq d = 2$ and $\mathcal{T}_1$ and $\mathcal{T}_2$ are independent and indistinguishable.

Fig. 8. Indistinguishable pairs of sets of interactions.

3.3. Properties of CLAs

The following observation follows from the definition.

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

Observation 2 states that when there are no constraints, an LA, if existing, and a CLA are equivalent.

Observation 2. *Suppose that the SUT has no constraints, i.e., $\phi(\boldsymbol{\sigma}) = true$ for all $\boldsymbol{\sigma} \in V_1 \times \cdots \times V_k$, and that an LA A exists (with any parameters of $0 \leq d < |S_{min}|, 0 \leq t \leq k$). 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.*

Proof. Suppose that A is a (d, t)-LA. Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{I}_t (= \mathcal{VI}_t)$ be any two interaction sets such that $|\mathcal{T}_1| = |\mathcal{T}_2| = d$. (1) If $\mathcal{T}_1 \neq \mathcal{T}_2$, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. If $\mathcal{T}_1 = \mathcal{T}_2$, then they are not distinguishable. Hence A is a (d, t)-CLA. (2) Suppose that an array A' is a (d, t)-CLA. If $\mathcal{T}_1 \neq \mathcal{T}_2$, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ and thus they are distinguishable, which in turn implies $\rho_{A'}(\mathcal{T}_1) \neq \rho_{A'}(\mathcal{T}_2)$. If $\mathcal{T}_1 = \mathcal{T}_2$, then they are not distinguishable and trivially $\rho_{A'}(\mathcal{T}_1) = \rho_{A'}(\mathcal{T}_2)$. Hence A' is a (d, t)-LA. The same argument applies to the other three types of LAs. □

It should be noted that LAs do not always exist and that the above observation claims the equivalence of an LA and a CLA only if the LA actually exists. On the other hand, CLAs always exist whether there are constraints or not, as will be shown in Theorem 1. For example, no (2, 1)-LAs exist for the running example: Consider $\mathcal{T}_1 = \{\{(4, 0)\}, \{(4, 1)\}\}$ and $\mathcal{T}_2 = \{\{(5, 0)\}, \{(5, 1)\}\}$. Then $\rho_A(\mathcal{T}_1)$ and $\rho_A(\mathcal{T}_2)$ both include all rows; thus $\rho_A(\mathcal{T}_1) = \rho_A(\mathcal{T}_2)$ for any A. The guarantee of the existence of CLAs comes from the definition which exempts indistinguishable pairs of interaction sets from fault localization. The example above illustrates that when $d \geq |S_{min}| = |S_4| = 2$. The definition of (2, 1)-LAs does not hold for any $N \times k$ array for the SUT. Because $\mathcal{T}_1$ and $\mathcal{T}_2$ are always indistinguishable and the existence of indistinguishable interaction set pairs violates the definition of (2, 1)-LAs. In contrast, CLAs permit the existence of indistinguishable interaction set pairs, and only distinguish interaction pairs that are distinguishable. Hence, (2, 1)-CLAs exist for the SUT while (2, 1)-LAs do not exist. In that sense, CLAs can be viewed as a "best effort" variant of LAs.

Lemma 1. *A pair of sets of valid interactions, $\mathcal{T}_1$ and $\mathcal{T}_2$, are distinguishable iff there is a valid test 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 $\boldsymbol{\sigma} \in \mathcal{R}$, $(\exists T \in \mathcal{T}_1 : T \subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma})$ 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 $\boldsymbol{\sigma}$. 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, i.e., for every valid test $\boldsymbol{\sigma}$, $(\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma}) \vee (\exists T \in \mathcal{T}_2 : T \subseteq \boldsymbol{\sigma})$ 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 $\boldsymbol{\sigma}$, $(\forall T \in \mathcal{T}_1 : T \not\subseteq \boldsymbol{\sigma}) \wedge (\forall T \in \mathcal{T}_2 : T \not\subseteq \boldsymbol{\sigma})$ 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 $\boldsymbol{\sigma}$ 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 1. *If A is an array consisting of all valid tests, then A is a (d, t)-, $(d, \overline{t})$-, $(\overline{d}, t)$- and $(\overline{d}, \overline{t})$-CLA for any $d \geq 0$ and $0 \leq t \leq k$.*

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

Although Theorem 1 guarantees that a test suite consisting of all valid tests is a CLA, it is desirable to use a smaller test suite

in practice. In Section 4, we present a computational method for generating small CLAs.

4. Computational generation of $(\overline{1}, \overline{t})$-CLAs

In this and next sections, we focus our attention on generation of $(\overline{1}, \overline{t})$-CLAs for practical reasons as follows. As demonstrated in the previous section, when the value of d (or $\overline{d}$) exceeds one, the size of CLAs may become substantially larger than t-CCAs, offsetting the very benefit of combinatorial interaction testing. Also, practical test suites must distinguish the situation where no fault exists from that where some hypothesized fault occurs; thus we consider $(\overline{1}, \overline{t})$-CLAs, instead of $(1, \overline{t})$-CLAs.

In this section, we propose an algorithm for generating $(\overline{1}, \overline{t})$-CLAs. Although not much research exists on generation of LAs, there has already been a large body of research on CCA generation in the combinatorial interaction testing field. The idea of the proposed algorithm is to make use of an existing CCA generation algorithm to generate $(\overline{1}, \overline{t})$-CLAs. This becomes possible by the theorem shown in Section 4.1, which proves that any $(t+1)$-CCA is a $(\overline{1}, \overline{t})$-CLA. This result allows us to use a two-step approach as follows:

Step 1. A $(t+1)$-CCA is generated using an off-the-shelf algorithm.

Step 2. A $(\overline{1}, \overline{t})$-CLA is obtained by removing redundant tests from the $(t+1)$-CCA.

4.1. Theoretical results

Theorem 2. *Let t be an integer such that $0 \le t < k$. If an $N \times k$ array A is a $(t+1)$-CCA, then A is also a $(\overline{1}, \overline{t})$-CLA.*

Proof. By Definition 2, 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 $0 \le |\mathcal{T}_1| \le 1$, $0 \le |\mathcal{T}_2| \le 1$, and $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinguishable. Now suppose that an $N \times k$ array A is a $(t+1)$-CCA such that $0 \le t < k$. 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. Since $\rho_A(\emptyset) = \emptyset$, $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2) = \emptyset$ holds for any $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ if $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 0$. The same argument clearly holds if $|\mathcal{T}_1| = 0$ and $|\mathcal{T}_2| = 1$.

In the rest of the proof, we consider the case in which $|\mathcal{T}_1| = 1$ and $|\mathcal{T}_2| = 1$. We 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 $T_b = \{(F_{b_1}, v_{b_1}), \ldots, (F_{b_m}, v_{b_m})\}$ $(0 \le l, m \le t)$. 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. Since 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. Since $\{T_a\}$ and $\{T_b\}$ are distinguishable, there must be at least one valid test $\boldsymbol{\sigma}$ 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 w_j, is different from v_j, because otherwise T_b were covered by $\boldsymbol{\sigma}$. Now consider a $(l+1)$-way interaction $T'_a = T_a \cup \{(F_j, w_j)\}$. Since the valid test $\boldsymbol{\sigma}$ covers T'_a, T'_a is a $(l+1)$-way valid interaction. Since A is a $(t+1)$-CCA and $l+1 \le t+1$, 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 $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{VI}_t}$ if $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$ and they are distinguishable. □

This theorem, namely, Theorem 2 can be viewed as a variant of Theorem 8.5 of Colbourn and McClary (2008), where it is proved, among other things, that a $(d+t)$-CA is a $(\overline{d}, t)$-LA. Theorem 2 shows that a $(t+1)$-CCA is already a $(\overline{1}, \overline{t})$-CLA. However, $(t+1)$-CCAs usually contain tests that are not needed to form $(\overline{1}, \overline{t})$-CLAs; thus such redundant tests should be removed to obtain small $(\overline{1}, \overline{t})$-CLAs.

Theorem 3 below proves that a $(\overline{1}, t)$-CLA and a $(\overline{1}, \overline{t})$-CLA are equivalent. This property is useful for checking whether the test is redundant or not. With this property, a test can be determined to be redundant if its removal does not invalidate the condition required for the array to be a $(\overline{1}, t)$-CLA, instead of a $(\overline{1}, \overline{t})$-CLA. This simplifies the check because we can restrict the interactions to be considered to those in $\mathcal{VI}_t$, instead of $\overline{\mathcal{VI}_t}$.

Lemma 2. *Suppose that an $N \times k$ array A is a $(\overline{1}, t)$-CLA such that $1 \le t \le k$. Then A is a t-CCA.*

Proof. Since 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 $|\mathcal{T}_1|, |\mathcal{T}_2| \le 1$. 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 3. *If an $N \times k$ array A is a $(\overline{1}, t)$-CLA such that $1 \le t \le k$, then A is a $(\overline{1}, \overline{t})$-CLA.*

Proof. Suppose that A is a $(\overline{1}, t)$-CLA such that $1 \le t \le k$. By Lemma 2, A is a t-CCA; thus, by Theorem 2, it is a $(\overline{1}, \overline{t-1})$-CLA. Recall that A is a $(\overline{1}, \overline{t})$-CLA iff $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ for all $\mathcal{T}_1, \mathcal{T}_2 \in \overline{\mathcal{VI}_t}$ such that $\mathcal{T}_1$ and $\mathcal{T}_2$ are distinguishable and $0 \le |\mathcal{T}_1|, |\mathcal{T}_2| \le 1$. (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 $T \in \overline{\mathcal{VI}_t}$. Since 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 of the proof, we assume that $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$. Let $\mathcal{T}_1 = \{T_a\}$, $\mathcal{T}_2 = \{T_b\}$ where $T_a, T_b \in \overline{\mathcal{VI}_t}$. Without losing generality, we assume that the strength of T_a is at most equal to that of T_b, i.e., $0 \le |T_a| \le |T_b| \le t$. If $0 \le |T_a| \le |T_b| \le t-1$ and $\{T_a\}$ and $\{T_b\}$ are distinguishable, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ since A is a $(\overline{1}, \overline{t-1})$-CLA. If $|T_a| = |T_b| = t$ and $\{T_a\}$ and $\{T_b\}$ are distinguishable, then $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ since A is a $(\overline{1}, t)$-CLA.

Now consider the remaining case where $0 \le |T_a| < |T_b| = t$. Assume that $\{T_a\}$ and $\{T_b\}$ are distinguishable. Below we show that $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$ under this assumption. Because of the assumption, at least either one of the following two cases holds: Case 1: for some $\boldsymbol{\sigma} \in \mathcal{R}$, $T_a \subseteq \sigma$ and $T_b \nsubseteq \boldsymbol{\sigma}$, or Case 2: for some $\boldsymbol{\sigma} \in \mathcal{R}$, $T_a \nsubseteq \boldsymbol{\sigma}$ and $T_b \subseteq \boldsymbol{\sigma}$.

Let $T_a = \{(F_{a_1}, u_{a_1}), \ldots, (F_{a_l}, u_{a_l})\}$ and $T_b = \{(F_{b_1}, v_{b_1}), \ldots, (F_{b_t}, v_{b_t})\}$ $(0 \le l \le t-1)$. 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 in $\mathcal{R}$ such that $T_a \subseteq \boldsymbol{\sigma}_1$ and $T_b \nsubseteq \boldsymbol{\sigma}_1$. Choose a factor F_{b_i}, $1 \le i \le t$ 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 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 $\boldsymbol{\sigma} \in \mathcal{R}$, $\hat{T} \subseteq \boldsymbol{\sigma} \Rightarrow T_b \nsubseteq \boldsymbol{\sigma}$ holds because $w_{b_i} \neq v_{b_i}$. Since A is a t-CCA and the strength of $\hat{T}$ is at

Algorithm 1: Algorithm for CLA generation

Input: SUT $\mathcal{M}$, strength t
Output: $(\overline{1}, \overline{t})$-CLA A

1 $A \leftarrow$ GENERATECCA($\mathcal{M}, t+1$)
  // generate (t+1)-CCA
2 $\mathcal{VI}_t \leftarrow$ GETALLINTERACTIONS(A, t)
  // get all t-way interactions from the (t+1)-CCA
3 $map \leftarrow$ MAPINTERACTIONTOROWS($\mathcal{VI}_t, A$)
  // get a mapping that maps $T \in \mathcal{VI}_t$ to a set of rows $\rho_A(T)$

4 **for** each *row* $\boldsymbol{\sigma} \in A$ **do**
  // randomly pick a row that has yet to be selected
5 $map' \leftarrow$ UPDATEMAP($map, \boldsymbol{\sigma}$)
6 // get a mapping for the array with $\boldsymbol{\sigma}$ removed
  $\mathcal{I} \leftarrow$ GETINTERACTIONS($\boldsymbol{\sigma}, t$)
7 // get all t-way interactions that appear in $\boldsymbol{\sigma}$
  if $(\forall T \in \mathcal{I} : map'(T) \neq \emptyset) \wedge$
8 $(\forall T_a \in \mathcal{I}, \forall T_b \in \mathcal{VI}_t : map(T_a) \neq map(T_b) \Rightarrow$
  $map'(T_a) \neq map'(T_b))$ **then**
    // test $\boldsymbol{\sigma}$ is redundant
9 $A \leftarrow A$ with $\boldsymbol{\sigma}$ removed
10 $map \leftarrow map'$

11 **return** A

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 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 $\check{T} \not\subseteq \boldsymbol{\sigma}_2$ (since $T_a \not\subseteq \boldsymbol{\sigma}_2$ 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 we can show that T_a and T_b never appear simultaneously in any test $\boldsymbol{\sigma} \in \mathcal{R}$ as follows. If there is some test $\boldsymbol{\sigma}$ in $\mathcal{R}$ in which T_a and T_b are both covered, then $\check{T}$ is also covered by some tests (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 in $\mathcal{R}$ that covers T_a and T_b. Since 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 $\rho_A(\mathcal{T}_1) \neq \rho_A(\mathcal{T}_2)$. □

4.2. Algorithm

Algorithm 1 is the proposed algorithm for generating $(\overline{1}, \overline{t})$-CLAs. The algorithm takes an SUT model $\mathcal{M}$ and strength t as input and finally returns a $(\overline{1}, \overline{t})$-CLA A. 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 $(t+1)$-CCA. Then the function GETALLINTERACTIONS() is called to enumerate all t-way interactions the $(t+1)$-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 $(t+1)$-CCA contains all t-way valid interactions. Once all the valid t-way interactions have been collected, we compute a mapping *map* that maps each of them to the set of rows of A that cover it; that is, $map : T \mapsto \rho_A(T)$ where $T \in \mathcal{VI}_t$.

In each iteration of the for loop, a row $\boldsymbol{\sigma}$ is randomly chosen from A. Then we compute map' which is a mapping such that $map' : T \mapsto \rho_A(T) \backslash \{\boldsymbol{\sigma}\}$. In other words, map' is $\rho_{A'}(T)$ where A' is the array obtained from A by removing $\boldsymbol{\sigma}$ from it. The function UPDATEMAP() is used to obtain map'. Also we enumerate all t-way interactions that are covered by $\boldsymbol{\sigma}$. The set of these interactions is represented by $\mathcal{I}$.

In each iteration of the loop, we check whether $\boldsymbol{\sigma}$ can be removed or not. The row can be removed if A remains to be a $(\overline{1}, t)$-CLA (equivalently, $(\overline{1}, \overline{t})$-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., $map'(T) \neq \emptyset$. The condition holds if and only if $\rho_{A'}(\mathcal{T}_1) \neq \rho_{A'}(\mathcal{T}_2)$ holds when $|\mathcal{T}_1| = 0$ and $|\mathcal{T}_2| = 1$, since $|\mathcal{T}_1| = 0$ implies $\mathcal{T}_1 = \emptyset$ which in turn implies $\rho_{A'}(\mathcal{T}_1) = \emptyset$.

The other condition corresponds to the case $|\mathcal{T}_1| = |\mathcal{T}_2| = 1$: 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 $\{T_a\}$ and $\{T_b\}$ are distinguishable, then $map'(T_a) \neq map'(T_b)$ (i.e., $\rho_{A'}(T_a) \neq \rho_{A'}(T_b)$). Note that $\{T_a\}$ and $\{T_b\}$ are distinguishable iff $map(T_a) \neq map(T_b)$ (i.e., $\rho_A(T_a) \neq \rho_A(T_b)$), since A is a $(\overline{1}, \overline{t})$-CLA.

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 of the first condition can be performed by examining only the interactions covered by $\boldsymbol{\sigma}$, i.e., those in $\mathcal{I}$, instead of all interactions in $\mathcal{VI}_t$. The same is true for checking of the second condition: it can be performed by checking each pair of an interaction T_a in $\mathcal{I}$ and another interaction $T_b \in \mathcal{VI}_t$.

The loop is iterated until all rows in the initial A have been examined. Finally, the resulting A becomes a $(\overline{1}, \overline{t})$-CLA of reduced size.

As stated above, output $(\overline{1}, \overline{t})$-CLAs vary for different runs of the algorithm, even if the initial A (i.e., the $(t+1)$-CCA generated in line 1) is identical for all runs. This is because the $(\overline{1}, \overline{t})$-CLAs finally obtained depends also on the order of deleting rows.

For example, suppose that there are only three valid t-way interactions T_1, T_2 and T_3 and that $\{T_1\}, \{T_2\}, \{T_3\}$ are distinguishable with each other. Also suppose that after mapping each interaction to rows, we have $map(T_1) = \{1, 2, 3\}$, $map(T_2) = \{1, 2, 4\}$ and $map(T_3) = \{4, 5\}$. If the order of deleting rows is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, rows 1 and 2 are deleted but rows 3, 4 and 5 are not. This is because after deleting rows 1 and 2, the mapping becomes: $map(T_1) = \{3\}$, $map(T_2) = \{4\}$ and $map(T_3) = \{4, 5\}$; thus any further deletion of rows would make some interaction lose all its covering rows or make identical the sets of covering rows for some pair of interactions. However, if the deleting order is $5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$, rows 5, 3 and 2 are deleted. The deleting order of rows thus influences the sizes of resulting CLAs.

Fig. 9 shows concrete examples of different CLAs obtained from the same CCA. Specifically, the two $(\overline{1}, \overline{1})$-CLAs in this figure were generated from the 2-CCA shown in Fig. 3. For the CLA on the left, the deleting order was $\underline{\boldsymbol{\sigma}_8} \rightarrow \boldsymbol{\sigma}_1 \rightarrow \underline{\boldsymbol{\sigma}_{11}} \rightarrow \boldsymbol{\sigma}_9 \rightarrow \underline{\boldsymbol{\sigma}_7} \rightarrow \boldsymbol{\sigma}_3 \rightarrow \boldsymbol{\sigma}_{10} \rightarrow \underline{\boldsymbol{\sigma}_6} \rightarrow \boldsymbol{\sigma}_5 \rightarrow \underline{\boldsymbol{\sigma}_2} \rightarrow \boldsymbol{\sigma}_4$, whereas the order was $\underline{\boldsymbol{\sigma}_3} \rightarrow \underline{\boldsymbol{\sigma}_6} \rightarrow \underline{\boldsymbol{\sigma}_5} \rightarrow \boldsymbol{\sigma}_4 \rightarrow \underline{\boldsymbol{\sigma}_{10}} \rightarrow \boldsymbol{\sigma}_9 \rightarrow \underline{\boldsymbol{\sigma}_8} \rightarrow \underline{\boldsymbol{\sigma}_2} \rightarrow \boldsymbol{\sigma}_{11} \rightarrow \boldsymbol{\sigma}_7 \rightarrow \boldsymbol{\sigma}_1$ for the CLA on the right. (The rows deleted are underlined.)

5. Evaluation

5.1. Experiment 1: Generation of CLAs with strength $t = 2$

In this section, the proposed generation algorithm is evaluated. Here, we focus on the case $t = 2$, i.e., the generation of $(\overline{1}, \overline{2})$-CLAs. The evaluation is performed with respect to two criteria:

σ_1	0	0	1	0	1
σ_3	0	1	1	0	0
σ_4	0	2	0	0	0
σ_5	0	2	0	1	1
σ_9	1	2	2	1	1
σ_{10}	2	1	2	1	1

σ_1	0	0	1	0	1
σ_4	0	2	0	0	0
σ_7	1	1	1	0	0
σ_9	1	2	2	1	1
σ_{11}	2	2	1	1	1

Fig. 9. Two $(\overline{1}, \overline{1})$-CLAs obtained by the algorithm. Both CLAs are generated by removing redundant rows from the 2-CCA shown in Fig. 3.

generation time and sizes (the number of rows) of CLAs. For comparison, we choose a generation algorithm based on an SMT (Satisfiability Modulo Theories) solver which we have proposed in Jin et al. (2018), because, to our knowledge, there does not exist another method that generates CLAs.

5.1.1. SMT-based generation algorithm

The SMT-based generation algorithm can be regarded as an adaptation of constraint solving-based methods for generating CCAs (Nanba et al., 2012; Banbara et al., 2010) or LAs (Konishi et al., 2017, 2020a). In this algorithm, the necessary and sufficient conditions for the existence of a $(\overline{1}, t)$-CLA (which is equivalent to a $(\overline{1}, \bar{t})$-CLA) of a given size N are encoded into a conjunction of logic expressions. Then, the algorithm uses an SMT solver to find a satisfiable valuation of variables of the logic expressions. If a satisfiable valuation is found, then it can be interpreted as a CLA. On the other hand, if there is no satisfiable valuation, then the non-existence of a CLA of size N can be concluded.

In the encoding of the conditions of a CLA, each cell of the array is represented as a variable; thus the array is encoded as a set of $N \times k$ variables. According to the definition of $(\overline{1}, t)$-CLAs (see Definition 2), three sets of logic expressions are needed. One of the three sets enforces that all rows of the array satisfy all of the SUT constraints. Another one is used to guarantee that each valid t-way interaction is covered by at least one row. This ensures that for every $T \in \mathcal{VI}_t$, $\rho_A(\{T\}) \neq \emptyset = \rho_A(\emptyset)$. The last one enforces that for every pair of valid t-way interactions, $T_a, T_b \in \mathcal{VI}_t$, if $\{T_a\}$ and $\{T_b\}$ are mutually distinguishable, there is at least one row covering only one interaction of the pair, i.e., $\rho_A(T_a) \neq \rho_A(T_b)$.

In our experiments, if a $(\overline{1}, \overline{2})$-CLA is successfully generated within a timeout period, we will decrease N by 1 and repeats runs of the algorithm until the SMT solver proves the non-existence of CLAs of size N. If a run of the algorithm fails to terminate within the timeout period, the repetition is stopped.

5.1.2. Research questions and experiment settings

We pose several research questions as follows for better understanding of experimental results.

RQ 1. *How does the proposed algorithm perform with respect to generation time and sizes for generated CLAs?*

RQ 2. *How different is the performance between the proposed algorithm and the SMT-based algorithm?*

RQ 3. *Does the proposed algorithm scale to real-world problems?*

We performed experiments where we applied both algorithms to a total of 30 problem instances, numbered from 1 to 30. Benchmarks No.1–5 are provided as part of the CitLab tool (Gargantini and Vavassori, 2012). Benchmarks No.6–25 can be found in Segall et al. (2011). Large benchmarks, namely, benchmarks No.26–30 are taken from Cohen et al. (2008). For each problem instance the proposed algorithm was executed 10 times, as it is a nondeterministic algorithm. On the other hand, the SMT-based algorithm was run only once, since it is deterministic. The initial value of N for the SMT-based algorithm was set to the size of the smallest CLAs among those obtained by the 10 runs of the proposed algorithm. This favors the SMT-based algorithm, since it ensures that the output CLA of the SMT-based algorithm is never greater in size than those obtained by the proposed heuristic algorithm.

All the experiments were conducted on a machine with 3.2 GHz 8-Core Intel Xeon W CPU and 128 GB memory, running MacOS Mojave. We wrote a C++ program that implements the proposed algorithm. The CIT-BACH tool [1] was used as a 3-way CCA generator. The implementation of the SMT-based algorithm was done using C. The Yices SMT solver (Dutertre, 2014) was used in this implementation. The timeout period was set to 1 h for every run of both algorithms.

The results of the experiments are shown in Table 1. The two leftmost columns of the table show the benchmark IDs and names. The third and fourth columns show the number of factors and the number of valid two-way interactions for each benchmark. The fifth column, marked with an asterisk (*), shows the number of unordered pairs $T_a, T_b (\neq T_a) \in \mathcal{VI}_2$ such that $\{T_a\}$ and $\{T_b\}$ are indistinguishable.

The remaining part of the table is divided into two parts: one for the proposed algorithm and the other for the SMT-based generation algorithm. In the proposed algorithm part, the left three columns show the maximum, minimum, and average sizes of the generated $(\overline{1}, \overline{2})$-CLAs. In the column labeled "Average (3-CCA)", the figures in parentheses indicate the sizes of the 3-way CCAs generated by GENERATECCA() on Line 1 in Algorithm 1. The next three columns indicate the maximum, minimum, and average running times. The running time is the sum of the time used for generating 3-way CCAs and the time used for deleting redundant rows from those 3-way CCAs. The unit is seconds.

The two rightmost columns show the results of the SMT-based algorithm. They show, for each problem instance, the size of the smallest CLA obtained and the running time taken by the algorithm to produce that CLA. (Thus, the running time does not include the running time of runs with $N > N_{sm}$ and $N < N_{sm}$, where N is the given size of an array and N_{sm} denotes the size of the smallest CLA.) As stated above, the algorithm was iterated with decreasing N until it failed to solve the problem within the timeout period or proves the nonexistence of a CLA of size N. In the latter case, the CLA obtained in the immediately previous iteration is guaranteed to be optimal in size. The figures in bold font show the sizes of these optimal $(\overline{1}, \overline{2})$-CLAs. The "T.O." marks indicate that even the first iteration with the initial N was not completed because of timeout.

The experimental results of the comparison on generation time and CLA sizes between different strength t is shown in Table 2. The first column in the table shows the benchmark IDs. The detailed system structure can be found in Table 1. In addition to the generation experiments for $(\overline{1}, \overline{2})$-CLAs, we applied our proposed method for the generation of $(\overline{1}, \overline{3})$-CLAs. The columns labeled "t" indicate the strength t of the generated CLA. As same as Table 1, the columns labeled "$|\mathcal{VI}_t|$" show the numbers of valid t-way interactions that benchmarks have. The column labeled "*" show the number of unordered pairs of interactions which are mutually indistinguishable. The generation for both $(\overline{1}, \overline{2})$-CLAs and $(\overline{1}, \overline{3})$-CLAs are repeated 10 times. The average generation time and the average sizes of the generated CLAs are shown in columns "average – time" and "average – size", respectively. Note that all experiments are set a timeout period as 1 h. The experiment data with "T.O." indicates that the corresponding execution did not finish within 1 h. The experiment data with

[1] CIT-BACH: https://osdn.net/users/t-tutiya/pf/cit_bach/.

"–" indicates that the $(t+1)$-CCAs for the systems cannot be constructed because $t+1$ exceeds the number of parameters in the system.

5.1.3. Experimental results

Answer to RQ 1. The proposed heuristic algorithm was able to find CLAs for all the benchmarks. The running time was even less than one second for many of these. Except for the two largest problem instances, it was at most 90 s. The two exceptional instances are Apache and GCC, both having nearly 200 factors. Even for these large benchmarks, the algorithm terminated, successfully producing CLAs within the one hour time limit. The proposed algorithm was able to generate CLAs that are considerably smaller than the initial CCAs. The reduction rate varies for different problem instances; but it was greater than 50% for many of the problems. Even a more than five-fold reduction was observed for some benchmarks, namely, Insurance (No. 14), NetworkMgmt (No. 15), Services (No. 18), Storage4 (No. 22), and Storage5 (No. 23). In summary, the proposed heuristic algorithm is able to generate CLAs within a reasonable time unless the problem is not very large. The sizes of CLAs produced by the algorithm are substantially smaller than the initial 3-CCAs.

Answer to RQ 2. When comparing the running times between both algorithms, the proposed algorithm shows distinguishing results. For all benchmarks except Car, Movie, Concurrency, the proposed algorithm achieved orders of magnitude reduction. The SMT-based algorithm often timed out even for the benchmarks that the proposed algorithm solved in less than one second. The difference can be explained as follows. To generate a CLA, the SMT-based algorithm needs to solve a constraint satisfaction problem represented by logic expressions. This problem can be very difficult to solve, especially when the given number of rows, N, approaches to the lower limit of the size of CLAs. On the other hand, the proposed heuristic simply repeats the check-and-delete process until all rows are examined. In the experiments, as stated above, we set the initial N of the SMT-based algorithm to the size of the smallest CLA obtained by 10 runs of the proposed heuristic algorithm. Hence the sizes of the CLAs generated by the SMT-based algorithm were guaranteed not to exceed those generated by the proposed heuristic algorithm. The experimental results show that the SMT-based algorithm was often successful in further decreasing the sizes of CLAs by, typically, a few rows. This also suggests that the proposed algorithm rarely produces the minimum (optimal) CLAs. One possible reason for this is that 3-way CCAs generated by GENERATECCA() may not be a superset of any of the optimal CLAs. Another reason is that resulting CLAs depends on the order of deleting rows. As there are a number of deleting orders, it can be unlikely that the one that leads to the optimal CLA, if any, is selected. In summary, the proposed heuristic algorithm runs much faster than does the SMT-based algorithm. If the problem is small enough for the SMT-based algorithm to handle, the algorithm is superior in yielding small CLAs to the proposed heuristic algorithm.

Answer to RQ 3. As stated, the proposed algorithm was able to produce CLAs in very short time for many problem instances. Even for very large benchmarks, namely, Apache and GCC, it completed generation of CLAs within one hour. These benchmarks are model taken from the real-world applications. Hence we conclude that, although further improvement is still desirable, the proposed algorithm can scale to real-world problems.

5.2. Experiment 2: Generation of CLAs of strength $t > 2$

The results of Experiment 1 showed that the proposed CLA generation algorithm can scale to large problems when the strength t of CLAs is two. Now we examine its scalability with respect to strength by posing the following research question.

RQ 4. *How does the proposed algorithm perform when the strength t of CLAs is relatively large ($t > 2$)?*

We applied the proposed algorithm to 28 and 27 of the 30 benchmark problems to create $(\overline{1}, \overline{t})$-CLAs with strength $t = 3$ and $t = 4$, respectively. The three problems, namely No. 4, No. 5, and No. 19, are excluded because they consist only of three or four factors.

As in Experiment 1, we ran the proposed algorithm 10 times for each problem. Table 2 summarizes the results of this experiment, including those obtained for $t = 2$ in Experiment 1. As in Table 1, the columns marked with "$|\mathcal{VI}_t|$" and "*" show respectively the number of valid interactions of strength t and the number of pairs of indistinguishable valid interactions of strength t. The columns labeled with "average" show the running time of the proposed algorithm and the size of obtained CLAs averaged over 10 runs.

From Table 2, it is seen that the size of generated CLAs and the generation time increased exponentially when the strength increased. For all benchmarks, the speed of growth in size was much slower than the speed of growth in generation time. With the one hour timeout, there were eight benchmark problems for which our algorithm ran out of time while generating $(\overline{1}, \overline{3})$-CLAs. Our proposed algorithm also failed to generate $(\overline{1}, \overline{4})$-CLAs for 14 benchmarks.

For large problems, the algorithm already failed to enumerate valid interactions. For example, for No 26, No 28, and No. 30, enumeration was not completed when $t = 3$. Even when valid interactions have been enumerated, if the number of these interactions was large, CDA generation was not completed within the timeout period.

Answer to RQ 4. The proposed algorithm is able to handle high strength $t \geq 3$ when the problem is not large. To handle large problems, further improvement in algorithm performance is needed.

Finally we note an interesting finding about the number of valid interactions. For benchmark No. 8, the number of valid 4-way interactions is smaller than the number of valid 3-way interactions. This would never happen when there were no constraints. In such an exceptional case, higher strength does not necessarily mean larger computation time. In fact, for this problem, the computation time did not vary much for different strengths.

5.3. Experiment 3: Applying CLA-based test cases to actual programs

In the third experiment, we examine CLAs with respect to the capability of identifying faulty interactions induced by real software bugs. By definition, CLAs ensure that faulty interactions can be located if underlying assumptions hold. Specifically, $(\overline{1}, \overline{t})$-CLAs allow any faulty interaction to be located if the strength of the interaction is at most t and there is no other faulty interaction. However, these assumptions may not necessarily hold in reality. The aim of this experiment is to answer the following research question.

RQ 5. *Can CLAs 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 is as follows:

Step 1 Construct SUT models for applications under test.
Step 2 Seed bugs to the source code of the application programs to create a collection of faulty versions of the programs.
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.

Table 1
Experimental results that compare CLA sizes and running times between the proposed heuristic algorithm and the SMT-based algorithm.

No.	SUT	$\|\mathcal{F}\|$	$\|\mathcal{VI}_2\|$	*	Proposed method						SMT based method	
					$(\overline{1},\overline{2})$-CLA sizes			Time (s)			$(\overline{1},\overline{2})$-CLA	Time (s)
					Max	Min	Average(3-CCA)	Max	Min	Average		
1	Aircraft	11	180	54	17	15	16.3 (23.6)	0.28	0.13	0.15	13	25.60
2	Car	9	102	161	**10**	**10**	**10.0** (12.2)	0.13	0.12	0.12	**10**	0.07
3	Movie	13	178	567	9	**8**	8.4 (11.2)	0.14	0.14	0.14	**8**	0.07
4	Medicitries	3	41	0	25	23	24.3 (41.0)	0.11	0.10	0.10	23	0.98
5	Medicitries_small	3	58	1	35	32	33.4 (70.0)	0.12	0.10	0.11	30	47.70
6	Banking1	5	102	0	28	25	26.7 (61.3)	0.14	0.13	0.14	23	913.41
7	Banking2	15	473	0	30	27	27.5 (41.2)	0.17	0.17	0.17	T.O.	T.O.
8	Concurrency	5	36	16	**7**	**7**	**7.0** (8.0)	0.11	0.10	0.10	**7**	0.01
9	CommProtocol	11	285	69	35	33	34.0 (54.6)	0.19	0.18	0.18	T.O.	T.O.
10	Healthcare1	10	361	5	50	45	48.2 (125.1)	0.21	0.19	0.20	42	3438.74
11	Healthcare2	12	466	0	36	33	34.9 (76.8)	0.21	0.20	0.20	T.O.	T.O.
12	Healthcare3	29	3092	477	101	77	91.9 (251.4)	4.59	4.03	4.38	T.O.	T.O.
13	Healthcare4	35	5707	288	105	98	101.7 (379.3)	21.59	20.34	20.85	T.O.	T.O.
14	Insurance	14	4573	0	805	789	794.2 (7325.3)	84.94	83.63	84.34	T.O.	T.O.
15	NetworkMgmt	9	1228	0	210	202	207.2 (1199.8)	1.89	1.84	1.86	T.O.	T.O.
16	ProcessorComm1	15	1058	6	63	58	59.9 (164.6)	0.62	0.56	0.59	T.O.	T.O.
17	ProcessorComm2	25	2525	1562	68	65	66.8 (200.3)	2.14	1.99	2.04	T.O.	T.O.
18	Services	13	1819	93	200	194	197.0 (1258.6)	4.70	4.52	4.62	T.O.	T.O.
19	Storage1	4	53	11	22	22	22.0 (25.0)	0.12	0.11	0.11	22	70.63
20	Storage2	5	126	0	37	34	35.9 (78.2)	0.12	0.11	0.12	30	564.92
21	Storage3	15	1020	57	89	87	87.5 (269.4)	0.81	0.75	0.77	T.O.	T.O.
22	Storage4	20	3491	0	222	215	218.6 (1183.1)	15.57	14.92	15.30	T.O.	T.O.
23	Storage5	23	5342	20	361	344	355.7 (2137.5)	56.71	55.82	56.18	T.O.	T.O.
24	SystemMgmt	10	310	130	31	27	29.0 (66.8)	0.17	0.15	0.16	T.O.	T.O.
25	Telecom	10	440	23	54	49	51.5 (144.5)	0.24	0.22	0.22	T.O.	T.O.
26	Apache	172	66,927	0	89	85	86.6 (232.4)	3350.32	3138.48	3263.15	T.O.	T.O.
27	Bugzilla	52	5818	0	48	41	45.3 (80.8)	9.80	7.98	9.02	T.O.	T.O.
28	GCC	199	82,770	46	68	61	64.4 (128.8)	2618.84	2286.57	2494.59	T.O.	T.O.
29	Spins	18	979	9	53	49	50.7 (136.3)	0.64	0.61	0.63	T.O.	T.O.
30	Spinv	55	8741	599	97	91	93.2 (321.4)	59.99	56.37	58.21	T.O.	T.O.

Step 4 Use CLAs to select test cases and locate (or estimate) faulty interactions using these test cases.
Step 5 Compare the results obtained from CLAs with the correct answers.

In the experiments, we set the parameters of CLAs, i.e., d and t, as $d = 1$ and $t = 2$.

5.3.1. Experimental setting

We chose *Flex* [2] and *Gzip* [3] as applications under test and obtained their source code from *Software-artifact Infrastructure Repository (SIR)* (Do et al., 2005) 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* (Ostrand and Balcer, 1988). Test specification files describe all of 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 including 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 (Petke et al., 2015). We used their SUT models in this experiment.

We used the bug seeding facility provided by SIR to seed bugs into *Flex* and *Gzip*. Exactly one bug was seeded in a single version of the programs.

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

From the test outcome, faulty interactions were identified as follows. We computed a *minimal* set of interactions such that (1) every interaction in the set occurs in some of the failed tests but not in any of the passed tests and (2) every failed test contains at least one interaction in the set. Here we say that the set is minimal if no smaller set satisfies these conditions. In general, there can be more than one such 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, no test case failed for some of the faulty versions. The bugs that manifested themselves are summarized in Table 3. The names of the bugs are designated by SIR.

Then, we ran our proposed algorithm to generate a $(\overline{1},\overline{2})$-CLA for the SUT model. We derived test scripts from the CLA and applied them to the set of faulty programs. The located faulty interactions using the CLA-based test cases are compared with the results of the exhaustive testing.

5.3.2. Experimental results

The results of the experiments are summarized in Table 4. The two leftmost columns show the applications and names of bugs. The rest of the table is divided into two parts, i.e., the exhaustive testing part and the CLA part. Each part consists of three columns. The "#Tests" column shows the total number of test cases. The "#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 correct answers.

For two faulty versions denoted 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 $(\overline{1},\overline{2})$-CLAs successfully identified the faulty interaction, as proved by the theory.

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

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

Table 2
Experimental results that compare average $(\overline{1}, \overline{t})$-CLA sizes and running times of the proposed algorithm with different strength t.

No.	t	$\lvert\mathcal{VI}_t\rvert$	*	Average		No.	t	$\lvert\mathcal{VI}_t\rvert$	*	Average	
				Time	Size					Time	Size
	2	180	54	0.15	16.3		2	1058	6	0.59	59.9
1	3	961	438	0.29	39.3	16	3	14,229	231	81.57	279.1
	4	3376	2109	1.25	85.1		4	130,725	4023	T.O.	T.O.
	2	102	161	0.12	10.0		2	2525	1562	2.04	66.8
2	3	346	1590	0.16	13.0	17	3	53,228	67,926	738.57	332.0
	4	701	6373	0.23	13.0		4	T.O.	T.O.	T.O.	T.O.
	2	178	567	0.14	8.4		2	1819	93	4.62	197.0
3	3	934	7489	0.28	13.0	18	3	30,031	4313	1278.08	1626.2
	4	3228	56,638	1.35	20.8		4	T.O.	T.O.	T.O.	T.O.
	2	41	0	0.10	24.3		2	53	11	0.11	22.0
4	3	–	–	–	–	19	3	71	43	0.14	23.0
	4	–	–	–	–		4	–	–	–	–
	2	58	1	0.11	33.4		2	126	0	0.12	35.9
5	3	–	–	–	–	20	3	432	0	0.22	116.3
	4	–	–	–	–		4	729	0	0.31	304.5
	2	102	0	0.14	36.7		2	1020	57	0.77	87.5
6	3	324	0	0.18	81.2	21	3	11,840	1212	61.72	396.5
	4	513	104	0.21	176.4		4	89,623	13,982	3145.79	1417.4
	2	473	0	0.17	27.5		2	3491	0	15.30	218.6
7	3	4290	0	2.84	81.2	22	3	86,153	0	T.O.	T.O.
	4	26,728	0	109.00	199.3		4	T.O.	T.O.	T.O.	T.O.
	2	38	16	0.10	7.0		2	5342	20	56.18	355.7
8	3	55	90	0.10	8.0	23	3	157,950	1908	T.O.	T.O.
	4	35	46	0.10	8.0		4	T.O.	T.O.	T.O.	T.O.
	2	285	69	0.18	34.0		2	310	130	0.16	29.0
9	3	1650	1221	0.68	79.9	24	3	1982	1591	0.73	88.6
	4	5978	9338	3.90	147.0		4	7770	10,227	5.12	216.8
	2	361	5	0.20	48.2		2	440	23	0.22	51.5
10	3	2535	118	2.25	190.5	25	3	3431	225	3.76	208.7
	4	11,102	1151	27.32	567.6		4	16,841	1246	66.62	688.5
	2	466	0	0.20	34.9		2	66,927	0	3263.15	86.6
11	3	4076	6	4.78	127.4	26	3	T.O.	T.O.	T.O.	T.O.
	4	23,792	183	166.26	416.7		4	T.O.	T.O.	T.O.	T.O.
	2	3092	477	4.38	91.9		2	5818	0	9.02	45.3
12	3	74,274	18,460	3051.87	484.6	27	3	202,683	24	T.O.	T.O.
	4	T.O.	T.O.	T.O.	T.O.		4	T.O.	T.O.	T.O.	T.O.
	2	5707	288	20.85	101.7		2	82,770	46	2494.59	64.4
13	3	191,398	13,378	T.O.	T.O.	28	3	T.O.	T.O.	T.O.	T.O.
	4	T.O.	T.O.	T.O.	T.O.		4	T.O.	T.O.	T.O.	T.O.
	2	4573	0	84.34	794.2		2	979	9	0.63	50.7
14	3	T.O.	T.O.	T.O.	T.O.	29	3	12,835	355	86.76	220.0
	4	T.O.	T.O.	T.O.	T.O.		4	116,332	6436	T.O.	T.O.
	2	1228	0	1.86	207.2		2	8741	599	58.21	93.2
15	3	15,370	1	203.995	1664.6	30	3	T.O.	T.O.	T.O.	T.O.
	4	116,350	40	T.O.	T.O.		4	T.O.	T.O.	T.O.	T.O.

Table 3
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 −−"

For F_AA_2, there were two faulty interactions which both were of strength one, namely, {(*FastSwithT*, *FST*)} and {(*FastSwithT*, *AlterFast*)}. The CLA-based test cases failed to locate either of these faulty interactions. In this case, there was no single interaction that appeared in the 11 failed test cases but not in the remaining 21 passed test cases. However, if we assumed the existence of two faulty interactions of strength ≤ 2, the faulty interactions could be identified because no other interaction pairs coincide the test outcome. This suggests that even if faulty interactions cannot be exactly located, the test outcome obtained from CLAs may provide informative clues about them.

Similarly to F_AA_2, the case F_AA_6 also contained two faulty interactions; but in this case, one of the faulty interactions was of strength four. In this case, the CLA-based test cases correctly located one faulty interaction that is 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.

Table 4
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 = Equivalence Classes;
FastS = Fast Scanner, FS = Fast Scan, FullS = Full Scan;
Set_V = Set V Option; Set_4 = Set 4 Option.

The case FAULTY_F_KL_6 contained one faulty interaction whose strength is three. This means that by definition, the $(\overline{1}, \overline{2})$-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 this is useful for fault localization as it is a subset of the correct faulty interaction {(*SetV, On*), (*Set4, On*), (*FileType, ASCII*)}.

Although this experiment is limited in scale, we answer RQ 5 based on the results obtained so far as follows.

Answer to RQ 5. The test cases derived from $(\overline{1}, \overline{t})$-CLAs may fail to locate faulty interactions if there are more than one faulty interaction or faulty interactions have strength greater than t; but even in such cases, they still can provide information useful for localization of faulty interactions.

6. Related work

Constraint handling has been an important issue in combinatorial interaction testing, even before the name of this testing approach was coined. Early work includes, for example, Tatsumi (1987) and Cohen et al. (1997). Recent surveys of constraint handling in combinatorial interaction testing include (Wu et al., 2019a; Wu et al., 2019b; Ahmed et al., 2017). These surveys mention more than 100 research papers addressing this particular problem.

In contrast, research on LAs is still in an early stage (Colbourn and Syrotiuk, 2016). The notion of LAs was originally proposed by Colbourn and McClary (2008). Since then, some studies have been published that discuss mathematical properties of LAs or propose mathematical constructions of LAs. These studies include (Shi et al., 2012b; Tang et al., 2012; Colbourn et al., 2016; Colbourn and Fan, 2016; Shi et al., 2020). Some other studies proposed computational generation methods of LAs (Konishi et al., 2017, 2020a; Nagamoto et al., 2014; Konishi et al., 2020b; Seidel et al., 2018; Lanus et al., 2019). None of these previous studies consider constraints. Recent surveys on the state of locating array research and its applications can be found in Colbourn and Syrotiuk (2016, 2018).

Mathematical objects similar to LAs include *Detecting Arrays* (Colbourn and McClary, 2008; Shi et al., 2012a; Colbourn and Syrotiuk, 2019) and *Error Locating Arrays* (Martínez et al., 2010). To our knowledge, no attempts have been reported to incorporate constraints into these arrays, either.

We for the first time introduced the concept of CLA in Jin and Tsuchiya (2018a), which is a preprint of an early version of this paper. This paper extends the early version by incorporating our subsequent work (Jin and Tsuchiya, 2018b), where we showed the heuristic algorithm for obtaining CLAs for the first time. Originally we presented it as a method of generating $(\overline{1}, t)$-CLAs, instead of $(\overline{1}, \overline{t})$-CLAs. This paper extends (Jin and Tsuchiya, 2018b) by providing new theorems (namely, Theorems 2 and 3) to show that the algorithm can yield $(\overline{1}, \overline{t})$-CLAs and by providing more comprehensive experimental results using a new, faster implementation of the algorithm. The SMT-based algorithm, which was compared with the proposed algorithm in Section 5, was presented in Jin et al. (2018).

There are many studies that address finding faulty interactions without using the mathematical objects mentioned above. Many of these studies proposed adaptive testing strategies (Wang et al., 2010; Zhang and Zhang, 2011; Li et al., 2012; Arcaini et al., 2019; Bonn et al., 2019; Niu et al., 2020b). In an adaptive strategy, new test cases are interactively created and executed to gradually narrow down the candidates for faulty interactions. On the other hand, the CLA-based approach is non-adaptive. A main benefit of using non-adaptive approaches is that testing, which is often very time-consuming, can be performed in parallel. Identifying multiple faulty interactions with an adaptive approach is discussed in Niu et al. (2020a).

In Yilmaz et al. (2006), Shakya et al. (2012) and Nishiura et al. (2017), machine learning techniques are applied to the test outcome to identify suspicious faulty interactions. Practical issues with fault localization raising in industrial software development cycles are discussed in Fouché et al. (2009) and Blue et al. (2019).

In this paper, we did not discuss how to spot faulty program statements from identified faulty interactions. This important problem is addressed in, for example, Ghandehari et al. (2013), Ghandehari et al. (2020).

7. Conclusions

In this paper, we introduced the notion of Constrained Locating Arrays (CLA), which generalize locating arrays by incorporating constraints on test parameters into them. The extension allows locating arrays to be applied to testing of real-world systems which usually have such constraints. We proved some basic properties of CLAs and then presented a heuristic algorithm to generate $(\overline{1}, \overline{t})$-CLAs which can locate at most one faulty interaction. Experimental results using a number of practical problem instances showed that the proposed algorithm is able to construct CLAs with reasonable time.

Even when multiple faults exist, the test outcome of $(\overline{1}, \overline{t})$-CLAs provides useful clues for identifying faulty interactions, because candidates for faulty interactions can be narrowed down to the interactions occurring only in the failing tests. However, a care must be taken if the test outcome matches a single fault and a set of multiple faults simultaneously. For example, consider the $(\overline{1}, \overline{1})$-CLA on the left in Fig. 9. If a one-way interaction $\{(1, 0)\}$ is the only faulty interaction, then the failing tests are $\boldsymbol{\sigma}_1$, $\boldsymbol{\sigma}_3$,

σ_4, and σ_5. This test outcome is the same when two interactions $\{(3, 0)\}$ and $\{(4, 0)\}$ are faulty. If the possibility of multiple faults is taken into account, it is not possible to conclude that $\{(1, 0)\}$ is the faulty interaction in this case. Detecting arrays address this problem in the absence of constraints. One possible direction of future research is to adapt detecting arrays to the SUTs that have constraints. Our early attempt in this direction can be found in Jin et al. (2020).

Other future research directions include, for example, developing other algorithms for CLA generation and extending CLAs to handle invalid inputs for testing negative scenarios (Fögen and Lichter, 2019a,b).

CRediT authorship contribution statement

Hao Jin: Conceived and designed the analysis, Collected the data, Contributed data or analysis tools, Performed the analysis, Wrote the paper. **Tatsuhiro Tsuchiya:** Conceived and designed the analysis, Contributed data or analysis tools, Performed the analysis, Wrote the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Ahmed, B.S., Zamli, K.Z., Afzal, W., Bures, M., 2017. Constrained interaction testing: A systematic literature study. IEEE Access 5 (99), 1. http://dx.doi.org/10.1109/ACCESS.2017.2771562.

Arcaini, P., Gargantini, A., Radavelli, M., 2019. Efficient and guaranteed detection of t-way failure-inducing combinations. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 200–209. http://dx.doi.org/10.1109/ICSTW.2019.00054.

Banbara, M., Matsunaka, H., Tamura, N., Inoue, K., 2010. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In: Proc. of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. In: LPAR'10, Springer-Verlag, Berlin, Heidelberg, pp. 112–126.

Blue, D., Hicks, A., Rawlins, R., Tzoref-Brill, R., 2019. Practical fault localization with combinatorial test design. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 268–271. http://dx.doi.org/10.1109/ICSTW.2019.00063.

Bonn, J., Fögen, K., Lichter, H., 2019. A framework for automated combinatorial test generation, execution, and fault characterization. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 224–233. http://dx.doi.org/10.1109/ICSTW.2019.00057.

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23 (7), 437–444. http://dx.doi.org/10.1109/32.605761.

Cohen, M.B., Dwyer, M.B., Shi, J., 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. Softw. Eng. 34, 633–650. http://dx.doi.org/10.1109/TSE.2008.50.

Colbourn, C.J., 2004. Combinatorial aspects of covering arrays. Le Mat. 58, 121–167.

Colbourn, C.J., Fan, B., 2016. Locating one pairwise interaction: Three recursive constructions. J. Algebra Combin. Discrete Struct. Appl. 3 (3), 127–134.

Colbourn, C.J., Fan, B., Horsley, D., 2016. Disjoint spread systems and fault location. SIAM J. Discrete Math. 30 (4), 2011–2026. http://dx.doi.org/10.1137/16M1056390.

Colbourn, C.J., McClary, D.W., 2008. Locating and detecting arrays for interaction faults. Journal of Combinatorial Optimization 15 (1), 17–48. http://dx.doi.org/10.1007/s10878-007-9082-4.

Colbourn, C.J., Syrotiuk, V.R., 2016. Coverage, location, detection, and measurement. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 19–25. http://dx.doi.org/10.1109/ICSTW.2016.38.

Colbourn, C.J., Syrotiuk, V.R., 2018. On a combinatorial framework for fault characterization. Math. Comput. Sci. 12 (4), 429–451. http://dx.doi.org/10.1007/s11786-018-0385-x.

Colbourn, C.J., Syrotiuk, V.R., 2019. Detecting arrays for main effects. In: Ćirić, M., Droste, M., Pin, J.-É. (Eds.), Algebraic Informatics. Springer International Publishing, Cham, pp. 112–123.

Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.. Empir. Softw. Eng.: Int J. 10 (4), 405–435.

Dutertre, B., 2014. Yices 2.2. In: Biere, A., Bloem, R. (Eds.), Computer-Aided Verification (CAV'2014). In: Lecture Notes in Computer Science, vol. 8559, Springer, pp. 737–744.

Fögen, K., Lichter, H., 2019a. Combinatorial robustness testing with negative test cases. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 34–45. http://dx.doi.org/10.1109/QRS.2019.00018.

Fögen, K., Lichter, H., 2019b. Repairing over-constrained models for combinatorial robustness testing. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 177–184. http://dx.doi.org/10.1109/QRS-C.2019.00045.

Fouché, S., Cohen, M.B., Porter, A.A., 2009. Incremental covering array failure characterization in large configuration spaces. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009. pp. 177–188. http://dx.doi.org/10.1145/1572272.1572294.

Gargantini, A., Vavassori, P., 2012. Citlab: A laboratory for combinatorial interaction testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 559–568. http://dx.doi.org/10.1109/ICST.2012.141.

Ghandehari, L.S., Lei, Y., Kacker, R., Kuhn, R., Xie, T., Kung, D., 2020. A combinatorial testing-based approach to fault localization. IEEE Trans. Softw. Eng. 46 (06), 616–645. http://dx.doi.org/10.1109/TSE.2018.2865935.

Ghandehari, L.S., Lei, Y., Kung, D., Kacker, R., Kuhn, R., 2013. Fault localization based on failure-inducing combinations. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 168–177. http://dx.doi.org/10.1109/ISSRE.2013.6698916.

Grindal, M., Offutt, J., Andler, S.F., 2005. Combination testing strategies: A survey. Softw. Test. Verif. Reliab. 15 (3), 167–199. http://dx.doi.org/10.1002/stvr.319.

Jin, H., Kitamura, T., Choi, E.-H., Tsuchiya, T., 2018. A satisfiability-based approach to generation of constrained locating arrays. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops. pp. 285–294. http://dx.doi.org/10.1109/ICSTW.2018.00062.

Jin, H., Shi, C., Tsuchiya, T., 2020. Constrained detecting arrays for fault localization in combinatorial testing. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. In: SAC '20, Association for Computing Machinery, New York, NY, USA, pp. 1971–1978. http://dx.doi.org/10.1145/3341105.3373952.

Jin, H., Tsuchiya, T., 2018a. Constrained locating arrays for combinatorial interaction testing. CoRR abs/1801.06041v1 arXiv:1801.06041v1 URL http://arxiv.org/abs/1801.06041v1.

Jin, H., Tsuchiya, T., 2018b. Deriving fault locating test cases from constrained covering arrays. In: 2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC). pp. 233–240. http://dx.doi.org/10.1109/PRDC.2018.00044.

Konishi, T., Kojima, H., Nakagawa, H., Tsuchiya, T., 2017. Finding minimum locating arrays using a sat solver. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 276–277. http://dx.doi.org/10.1109/ICSTW.2017.49.

Konishi, T., Kojima, H., Nakagawa, H., Tsuchiya, T., 2020a. Finding minimum locating arrays using a csp solver. Fund. Inform. 174, 27–42. http://dx.doi.org/10.3233/FI-2020-1929.

Konishi, T., Kojima, H., Nakagawa, H., Tsuchiya, T., 2020b. Using simulated annealing for locating array construction. Inf. Softw. Technol. 126, 106346. http://dx.doi.org/10.1016/j.infsof.2020.106346.

Lanus, E., Colbourn, C.J., Montgomery, D.C., 2019. Partitioned search with column resampling for locating array construction. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 214–223. http://dx.doi.org/10.1109/ICSTW.2019.00056.

Li, J., Nie, C., Lei, Y., 2012. Improved delta debugging based on combinatorial testing. In: 2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012. pp. 102–105. http://dx.doi.org/10.1109/QSIC.2012.28.

Martínez, C., Moura, L., Panario, D., Stevens, B., 2010. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. SIAM J. Discrete Math. 23 (4), 1776–1799. http://dx.doi.org/10.1137/080730706.

Nagamoto, T., Kojima, H., Nakagawa, H., Tsuchiya, T., 2014. Locating a faulty interaction in pair-wise testing. In: Proc. IEEE 20th Pacific Rim International Symposium on Dependable Computing (PRDC 2014). pp. 155–156. http://dx.doi.org/10.1109/PRDC.2014.26.

Nanba, T., Tsuchiya, T., Kikuno, T., 2012. Using satisfiability solving for pairwise testing in the presence of constraints. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E95.A (9), 1501–1505. http://dx.doi.org/10.1587/transfun.E95.A.1501.

Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Comput. Surv. 43, 11:1–11:29. http://dx.doi.org/10.1145/1883612.1883618.

Nishiura, K., Choi, E., Mizuno, O., 2017. Improving faulty interaction localization using logistic regression. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 138–149. http://dx.doi.org/10.1109/QRS.2017.24.

Niu, X., Nie, C., Lei, J.Y., Leung, H., Wang, X., 2020a. Identifying failure-causing schemas in the presence of multiple faults. IEEE Trans. Softw. Eng. 46 (2), 141–162. http://dx.doi.org/10.1109/TSE.2018.2844259.

Niu, X., Nie, C., Leung, H., Lei, Y., Wang, X., Xu, J., Wang, Y., 2020b. An interleaving approach to combinatorial testing and failure-inducing interaction identification. IEEE Trans. Softw. Eng. 46 (6), 584–615. http://dx.doi.org/10.1109/TSE.2018.2865772.

Ostrand, T., Balcer, M., 1988. The category-partition method for specifying and generating functional tests.. Commun. ACM 31, 676–686. http://dx.doi.org/10.1145/62959.62964.

Petke, J., Cohen, M.B., Harman, M., Yoo, S., 2015. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Trans. Softw. Eng. 41 (9), 901–924. http://dx.doi.org/10.1109/TSE.2015.2421279.

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

Seidel, S.A., Sarkar, K., Colbourn, C.J., Syrotiuk, V.R., 2018. Separating interaction effects using locating and detecting arrays. In: Iliopoulos, C., Leong, H.W., Sung, W.-K. (Eds.), Combinatorial Algorithms. Springer International Publishing, Cham, pp. 349–360.

Shakya, K., Xie, T., Li, N., Lei, Y., Kacker, R., Kuhn, D.R., 2012. 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. pp. 620–623. http://dx.doi.org/10.1109/ICST.2012.149.

Shi, C., Jin, H., Tsuchiya, T., 2020. Locating arrays with mixed alphabet sizes. MDPI Math. 8 (5), 831.

Shi, C., Tang, Y., Yin, J., 2012a. The equivalence between optimal detecting arrays and super-simple OAs. Des. Codes Cryptogr. 62 (2), 131–142. http://dx.doi.org/10.1007/s10623-011-9498-9.

Shi, C., Tang, Y., Yin, J., 2012b. Optimal locating arrays for at most two faults. Sci. China Math. 55 (1), 197–206. http://dx.doi.org/10.1007/s11425-011-4307-5.

Tang, Y., Colbourn, C.J., Yin, J., 2012. Optimality and constructions of locating arrays. J. Stat. Theory Pract. 6 (1), 20–29. http://dx.doi.org/10.1080/15598608.2012.647484.

Tatsumi, K., 1987. Test case design support system. In: Proc. of International Conference on Quality Control (ICQC'87), pp. 615–620.

Wang, Z., Xu, B., Chen, L., Xu, L., 2010. Adaptive interaction fault location based on combinatorial testing. In: 2010 10th International Conference on Quality Software. pp. 495–502. http://dx.doi.org/10.1109/QSIC.2010.36.

Wu, H., Changhai, N., Petke, J., Jia, Y., Harman, M., 2019a. Comparative analysis of constraint handling techniques for constrained combinatorial testing. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2019.2955687, (early access).

Wu, H., Nie, C., Petke, J., Jia, Y., Harman, M., 2019b. A survey of constrained combinatorial testing. arXiv preprint arXiv:1908.02480.

Yilmaz, C., Cohen, M.B., Porter, A.A., 2006. Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans. Softw. Eng. 32 (1), 20–34. http://dx.doi.org/10.1109/TSE.2006.8.

Zhang, Z., Zhang, J., 2011. 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. pp. 331–341. http://dx.doi.org/10.1145/2001420.2001460.

Hao Jin is a doctoral student in Graduate School of Information Science and Technology of Osaka University. He received a Bachelor's degree from Dalian University of Technology in Software Engineering and Japanese Language in 2016 and received his Master's degree in Information System Engineering from Osaka University in 2019. He currently works on combinatorial testing design.

Tatsuhiro Tsuchiya is currently a professor at the Graduate School of Information Science and Technology at Osaka University. He received the M.E. and Ph.D. degrees from Osaka University in 1995 and 1998, respectively. His research interests are in the areas of model checking, software testing, and distributed fault-tolerant systems.