Information and Software Technology 62 (2015) 198–213

Contents lists available at ScienceDirect

Information and Software Technology

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

Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures

Changhai Nie [a,*], Huayao Wu [a], Xintao Niu [a], Fei-Ching Kuo [b], Hareton Leung [c], Charles J. Colbourn [d]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[b] Faculty of Information and Communication Technologies, Swinburne University of Technology, VIC, Australia
[c] Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
[d] Arizona State University, Tempe, AZ, USA

ARTICLE INFO

Article history:
Received 2 October 2014
Received in revised form 16 February 2015
Accepted 17 February 2015
Available online 3 March 2015

Keywords:
Software testing
Random Testing (RT)
Adaptive Random Testing (ART)
Combinatorial Testing (CT)
Interaction Triggered Failure (ITF)
Minimal Failure-causing Schema (MFS)

ABSTRACT

Context: Software behavior depends on many factors, and some failures occur only when certain factors interact. This is known as an interaction triggered failure, and the corresponding selection of factor values can be modeled as a Minimal Failure-causing Schema (MFS). (An MFS involving m factors is an m-MFS.) Combinatorial Testing (CT) has been developed to exercise ("hit") all MFS with few tests. Adaptive Random Resting (ART) endeavors to make tests as different as possible, ensuring that testing of MFS is not unnecessarily repeated. Random Testing (RT) chooses tests at random without regard to the MFS already treated. CT might be expected to improve on RT for finding interaction triggered faults, and yet some studies report no significant difference. CT can also be expected to be better than ART, and yet other studies report that ART can be much better than RT. In light of these, the relative merits of CT, ART, and RT for finding interaction triggered faults are unclear.

Objective: To investigate the relationships among CT, ART, and RT, we conduct the first complete and systematic comparison for the purpose of hitting MFS.

Method: A systematic review of six aspects of CT, RT and ART is conducted first. Then two kinds of experiments are used to compare them under four metrics.

Results: ART improves upon RT, but t-way CT is better than both. In hitting t'-MFS the advantage is typically in the range from 10% to 30% when $t = t'$, but becomes much smaller when $t' < t$, and there may be no advantage when $t' > t$. The latter case may explain the studies reporting no significant difference between RT and CT.

Conclusion: RT is easily implemented. However, depending on its implementation, ART can improve upon RT. CT does as well as ART whether or not$t' = t$, but provides a valuable improvement in the cases when $t' = t$.

1. Introduction

Behavior of modern software systems can be affected by many factors. These systems can be developed by many methodologies, such as object oriented programming, component based design, and service oriented architecture. When these software systems consist of many different objects, components and services, different parts may interact to cause a failure, called an *interaction triggered failure (ITF)*. An ITF can occur when certain inputs or features are tested together; the value combination of factors causing the failure can be modeled as a *Minimal Failure-causing Schema (MFS)*. An MFS is often hidden in software, especially when the software system has many inputs and features.

1.1. Background

Suppose that the Software Under Test (SUT) has n factors $\{c_i : 1 \leqslant i \leqslant n\}$. These may represent configuration parameters of the configurable system or the running environment, internal or external events, different objects, services, features, user inputs, and so on. Values of these factors, or interactions among their values, may influence the behavior of the SUT. Each factor c_i has a_i values, which form a value set V_i. Any of these factors may interact and cause some ITF. To detect an ITF, we design test cases. A test case is a tuple $t = (v_1, v_2, \ldots, v_n)$, with $t(i) = v_i$ and $v_i \in V_i$ for $1 \leqslant i \leqslant n$. Let T_{all} denote all the possible test cases for the SUT:

* Corresponding author.
E-mail address: changhainie@nju.edu.cn (C. Nie).

http://dx.doi.org/10.1016/j.infsof.2015.02.008
0950-5849/

$T_{all} = \{t = (v_1, v_2, \ldots, v_n), t(i) = v_i \text{ where } v_i \in V_i, (1 \leqslant i \leqslant n)\}$. The size $|T_{all}| = a_1 \times a_2 \cdots \times a_n$ is prohibitively large for most systems.

When an ITF is triggered by a specific value combination of k factors ($k \leqslant n$), the value combination can be presented as an n-tuple, $(v_1, \ldots, v_n)$ with $v_i \in V_i \cup \{-\}$ for $1 \leqslant i \leqslant n$. When for k distinct indices $K = \{n_1, \ldots, n_k\}$ with $1 \leqslant n_1 < n_2 < \cdots < n_k \leqslant n$, we find that $v_{n_i} \in V_{n_i}$ for $1 \leqslant i \leqslant k$ and $v_j = -$ when $j \notin K$, this is a *k-value schema*. When no assigned value can be changed to $-$ without failing to observe the ITF, the schema is the root cause of ITF, the *Minimal Failure-causing Schema (MFS)* or *k-MFS*. We have discussed the existence of MFS, their properties in an SUT, and given a methodology for catching MFS in [1]. An ITF can be detected by selecting test cases that include the corresponding MFS. For each test case $t = (v_1, v_2, \ldots, v_n)$ of SUT, t can cover many k-value schemas, formed by some k parameters (in total, $2^n - 1 = C_n^1 + C_n^2 + \cdots + C_n^{n-1} + C_n^n = \sum_{k=1}^{n} C_n^k$). Because the size of T_{all} for some SUT is very large, it is a challenge to hit all MFS.

A simple way to hit MFS of SUT is to sample test cases from T_{all} at random with uniform distribution, using Random Testing (RT). RT selects test cases at random without considering any execution or structure information. It is a widely used and important testing method, often used as a comparison baseline to assess other testing techniques.

When a test case of RT is executed but fails to find an ITF, no value combinations of this test case can reveal ITFs. With this in mind, when selecting the next test case for RT, we should ensure that the next test case covers more new combinations, so that we may have a higher chance to detect an ITF. This variant of RT is *Adaptive Random Testing (ART)*. The fundamental idea behind ART is to reward diversity among sampled test cases. If a test case does not detect a failure, it seems better to sample test cases that are "far away" from it.

ART can be used in all cases where RT applies. Research has reported that ART can improve RT greatly for many applications [2], but we have not seen any work in using ART to hit MFS. Does it perform better in hitting the MFS than RT? In this paper we will explore this and other issues.

Combinatorial Testing (CT) aims to detect the ITF in SUT systematically; the number of CT research papers has grown rapidly in the last two decades [3]. CT uses a mathematical object called a covering array as the test suite to examine as many parameter value combinations as possible. For an SUT, a *covering array* is an $m \times n$ matrix in which every combination of any τ parameter values in SUT must appear in at least one of the m rows. It is also called a *τ-way covering array*. Testing with a τ-way covering array is *τ-way CT*, or *pairwise testing* when $\tau = 2$.

CT, or *combinatorial interaction testing*, is a Design Of Experiments (DOE) approach [4]. It provides a natural sampling mechanism in a large combination space, providing a small test suite to cover all value combinations. Some research suggests that most software faults are triggered by no more than six interacting factors [5], so τ-way CT may be as effective as n-way CT (*exhaustive testing*) even when $\tau \ll n$.

One might expect CT to be better than RT in hitting MFS, because RT simply generates test cases randomly. Also CT ought to be better than ART because ART was not developed to target the MFS. However, some studies show that there is no significant difference between CT and RT [6–8], while other studies indicate that ART can detect a failure using up to 50% fewer test cases than RT [9,2]. These results suggest that ART could be better than CT, contrary to our intuition.

It has been argued that CT is over promoted and poorly understood [10]. Accordingly, its effectiveness could be an illusion, or we may have evaluated the failure detecting ability of CT improperly. Given these concerns, three key aspects may affect its effectiveness:

1. Can CT be applied? (Does the SUT have multiple factors, each with a number of values, whose interactions may affect software quality?)
2. Have the right factors been identified? Have the right values been identified for each factor?
3. Is there any interaction among factors evidencing an MFS?

Of course, these aspects apply equally well to RT and ART.

1.2. Research issues

In this paper, we explore the following questions:

1. What role does RT play in hitting the MFS? Is it useful, as reported by other studies? How many test cases does RT need to achieve the target combination coverage, how does it scale, and how likely is it to yield similar results if we re-run RT on an SUT that contains MFS?
2. It has been reported that there is no significant difference in the failure detecting effectiveness between CT and RT. Can we confirm this result from the perspective of the MFS?
3. ART has been proposed as an enhancement to RT, and some experimental studies show that ART can detect a failure using 50% fewer test cases. Do we get similar results when targeting the MFS specifically?
4. If there is no significant difference between RT and CT in failure detecting effectiveness, and ART can improve RT greatly, can we deduce that ART is better than CT? Is this supposition true when effectiveness is measured in terms of the MFS?
5. What is the advantage (if any) of CT compared to RT and ART? Can we rank CT, RT and ART in terms of effectiveness?

1.3. Contributions

The main contributions of this paper are:

- We conduct a systematic review of six aspects of CT, RT and ART, including concept, principle, procedure, examples, advantages and disadvantages, and related research areas, to make a thorough comparison.
- We propose a new metric, the rate of fault detection (RFD) for the MFS, define it formally, study its properties, and then use it to evaluate RT, ART and CT.
- We present two versions of CT: offline CT, using the smallest known covering array; and online CT using a greedy algorithm to generate a covering array which may not be the smallest. We show that, using the same size test suite, τ-way offline CT is better than online CT, ART and RT in hitting m-MFS when $m \leqslant \tau$, but we also show with experiments that it is not true when $m > \tau$.
- We apply the conventional ART(HD) with "Hamming Distance" to hit the MFS, and propose a new ART(SD) with "Schema Distance". Both can improve upon RT, but cannot be better than CT, in hitting the MFS. We conclude that ART can greatly enhance RT, but it depends on the implementation.
- We propose a formal quantitative framework for comparing CT, RT and ART, which includes the metrics: (a) the rate of the MFS detection (RFD), (b) combination coverage, (c) diversity of the test suites and (d) size of test suite to reach a fixed combination coverage.
- We conduct two kinds of experiments: (1) to generate test suites of the same size with CT, RT and ART, compare their τ-way coverage, τ-way diversity and τ-way RFD ($\tau = 2, 3, 4$), (2) to achieve an assigned 100% τ-way coverage ($\tau = 2, 3, 4$), compare the size of test suites needed by CT, RT and ART. While only τ-way CT has an obvious advantage in hitting τ-MFS, there is no significant difference in hitting m-MFS ($m \neq \tau$) between RT and ART.

1.4. Organization

The remainder of the paper is organized as follows. Section 2 presents formal definitions and gives some metrics for measuring quality of the test suite in hitting MFS. We also propose RFD for the MFS and study its properties. Section 3 undertakes a systematic review of RT, ART and CT, and defines algorithms for each to capture the MFS. Section 4 builds a framework and constructs experiments to compare RT, ART and CT. Section 5 analyzes and compares existing studies on CT, RT and ART with our work. In Section 6, we conclude and propose future work.

2. ITF and quality metrics for test suites

We first give the formal definition of ITF and MFS, then define three metrics for evaluating test suite quality: combination coverage, diversity, and RFD of the test suite used. A small test example SUT' is used to illustrate the definitions and show how CT, ART and RT work to detect ITF. SUT' has 4 parameters c_1, c_2, c_3 and c_4, and each parameter c_i has $a_i = 3$ values as shown in Table 1, that is, $a_1 = a_2 = a_3 = a_4 = 3$. An example test case t of SUT' is the vector (A_2, B_1, C_0, D_1) where $c_1 = t(1) = A_2, c_2 = t(2) = B_1, c_3 = t(3) = C_0$ and $c_4 = t(4) = D_1$. There are $3 \times 3 \times 3 \times 3 = 81$ test cases in total of SUT', that is, $|T_{all}| = 81$.

2.1. ITF and MFS

We introduce definitions and theorems, and explain their use on SUT'.

Definition 1 (*k-value schema*). For SUT, the n-tuple, $(v_1, \ldots, v_n)$ with $v_i \in V_i \cup \{-\}$ for $1 \leqslant i \leqslant n$ is a k-value schema when for k distinct indices $K = \{i_1, \ldots, i_k\}$ with $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$, we find that $v_{i_j} \in V_{i_j}$ for $1 \leqslant j \leqslant k$ and $v_j = -$ if $j \notin K$. (When $k = n$, the n-tuple is a test case.)

An interaction among parameters is the effect caused by the combination of their values. When specific values of k parameters together trigger a failure, we interpret that it is the k-value schema of the SUT causes an ITF.

For a k-value schema $(-, v_{i_1}, \ldots, v_{i_k}, \ldots)(k > 0)$, if there exists a test case $t \in T_{all}$, where $t(i_j) = v_{i_j}, 1 \leqslant j \leqslant k$, then t covers (hits, contains or includes) the k-value schema or the k-value schema is covered by (hit by, contained in, or included in) t; otherwise, t does not cover the k-value schema.

Each test case $t = (v_1, v_2, \ldots, v_n), t(i) = v_i$ and $v_i \in V_i, (1 \leqslant i \leqslant n)$ for the SUT covers C_n^k k-value schemas $(k > 0)$, for a total of $\sum_{k=1}^{n} C_n^k = 2^n - 1$ k-value schemas with $1 \leqslant k \leqslant n$.

Table 1
The four input parameters of SUT'.

Index	c_1	c_2	c_3	c_4
1	A_0	B_0	C_0	D_0
2	A_1	B_1	C_1	D_1
3	A_2	B_2	C_2	D_2

Table 2
$CA(9; 2, 3^4)$ by CT, $(T(CT))$.

Index	c_1	c_2	c_3	c_4
1	A_0	B_0	C_0	D_0
2	A_0	B_1	C_1	D_1
3	A_0	B_2	C_2	D_2
4	A_1	B_0	C_1	D_2
5	A_1	B_1	C_2	D_0
6	A_1	B_2	C_0	D_1
7	A_2	B_0	C_2	D_1
8	A_2	B_1	C_0	D_2
9	A_2	B_2	C_1	D_0

Definition 2 (*Parent and subschema*). Let s_l be an l-value schema, and s_m be an m-value schema for the SUT with $l \leqslant m$. If all the fixed parameter values in s_l are also in s_m, then s_m subsumes s_l, denoted by $s_l \prec s_m$. In this case s_l is a subschema of s_m and s_m is a parent-schema of s_l.

For example, the 2-value schema $s_2 = (-, B_1, -, D_0)$ of SUT' is a subschema of the 3-value schema $s_3 = (-, B_1, C_2, D_0)$, so $s_2 \prec s_3$.

Definition 3 (*ITF*). For SUT, an *interaction triggered failure (ITF)* is a failure triggered by any test case that covers some k-value schema s_k with $k > 1$, that is, if any test case t $(t \in T_{all})$ covers s_k, it triggers this failure. s_k is a failure-causing schema of the ITF. When $k = 1$, this is a parameter value triggered failure.

Definition 4 (*m-MFS [1]*). An m-value schema s_m of the SUT is a Minimal Failure-causing Schema (MFS) if all the test cases $t \in T_{all}$ containing the schema s_m trigger a software failure, and for all the subschemas s_l of s_m ($l < m$ and $s_l \prec s_m$), there exists a test case $t \in T_{all}$ that contains s_l, but fails to cause ITF. The MFS s_m is denoted as m-MFS, where m is the degree of the MFS. When a test case t contains a MFS, it hits the MFS.

By Definitions 3 and 4, for a MFS s_k, if s_m is its parent schema, that is, $s_k \prec s_m$, then s_m is a failure-causing schema of ITF. For example, in SUT', if $(-, B_2, -, D_1)$ is a MFS of the ITF, any test case, such as (A_1, B_2, C_1, D_1), which covers this MFS, must fail. Hence there exist many failure-causing schemas for a single ITF, but we need only consider MFS. The goal of catching ITF is to design a small test suite to cover as many k-value schemas as possible for SUT. We use the terms hitting MFS, catching ITF, detecting ITF, detecting MFS, and catching MFS interchangeably.

As testing effectiveness depends on the generated test suite, we propose several metrics to evaluate the quality of a test suite.

2.2. Combination coverage

A test suite to hit ITF can be evaluated by its combination coverage: the number of value schemas it can cover expressed as a percent of all the value schemas of SUT.

Definition 5 (*Combination coverage*). Let T={$t_1, t_2, \ldots, t_m$} be a test suite of SUT, $t_i(\tau\text{-}value)$ be the set of all the τ-*value* schemas covered by t_i, $T(\tau - value) = \bigcup_{i=1}^{m} t_i(\tau - value)$ be all the τ-*value* schemas that test suite T can cover, and $SUT(\tau\text{-}value)$ be the set of all the τ-*value* schemas of SUT needed to be covered. (The number of all τ-*value* schemas in *SUT* is $|SUT(\tau\text{-}value)| = \sum_{1 \leqslant i_1 < i_2 < \cdots < i_\tau \leqslant n} a_{i_1} a_{i_2}, \ldots, a_{i_\tau}$.) The τ-*way* combination coverage of T is defined as $\tau\text{-}cov(T) = \frac{|T(\tau\text{-}value)|}{|SUT(\tau\text{-}value)|}$. The total combination coverage of T is defined as total-$cov(T)$, which is the ratio of the number of all the value schemas covered by T, $|\bigcup_{\tau=1}^{n} T(\tau - value)| = \sum_{\tau=1}^{n} |T(\tau - value)|$, to the total number of value schemas of SUT,

$$\left|\bigcup_{\tau=1}^{n} SUT(\tau\text{-}value)\right| = \sum_{\tau=1}^{n} |SUT(\tau\text{-}value)| = \sum_{\tau=1}^{n} \sum_{1 \leqslant i_1 < i_2 < \cdots < i_\tau \leqslant n} a_{i_1} a_{i_2}, \ldots, a_{i_\tau}.$$

Table 3
2-Value schema covered by each test case.

$t_i, 1 \leqslant i \leqslant 8$	2-Value schemas covered by t_i
$t_1 = (A_2, B_1, C_0, D_1)$	$(A_2, B_1, -, -), (A_2, -, C_0, -), (A_2, -, -, D_1), (-, B_1, C_0, -)$,$(-, B_1, -, D_1), (-, -, C_0, D_1)$
$t_2 = (A_1, B_1, C_0, D_1)$	$(A_1, B_1, -, -), (A_1, -, C_0, -), (A_1, -, -, D_1), (-, B_1, C_0, -), (-, B_1, -, D_1), (-, -, C_0, D_1)$
$t_3 = (A_1, B_0, C_0, D_1)$	$(A_1, B_0, -, -), (A_1, -, C_0, -), (A_1, -, -, D_1), (-, B_0, C_0, -), (-, B_0, -, D_1), (-, -, C_0, D_1)$
$t_4 = (A_1, B_1, C_1, D_1)$	$(A_1, B_1, -, -), (A_1, -, C_1, -), (A_1, -, -, D_1), (-, B_1, C_1, -), (-, B_1, -, D_1), (-, -, C_1, D_1)$
$t_5 = (A_1, B_1, C_0, D_0)$	$(A_1, B_1, -, -), (A_1, -, C_0, -), (A_1, -, -, D_0), (-, B_1, C_0, -), (-, B_1, -, D_0), (-, -, C_0, D_0)$
$t_6 = (A_2, B_0, C_0, D_1)$	$(A_2, B_0, -, -), (A_2, -, C_0, -), (A_2, -, -, D_1), (-, B_0, C_0, -), (-, B_0, -, D_1), (-, -, C_0, D_1)$
$t_7 = (A_1, B_0, C_2, D_1)$	$(A_1, B_0, -, -), (A_1, -, C_2, -), (A_1, -, -, D_1), (-, B_0, C_2, -), (-, B_0, -, D_1), (-, -, C_2, D_1)$
$t_8 = (A_0, B_2, C_2, D_2)$	$(A_0, B_2, -, -), (A_0, -, C_2, -), (A_0, -, -, D_2), (-, B_2, C_2, -), (-, B_2, -, D_2), (-, -, C_2, D_2)$

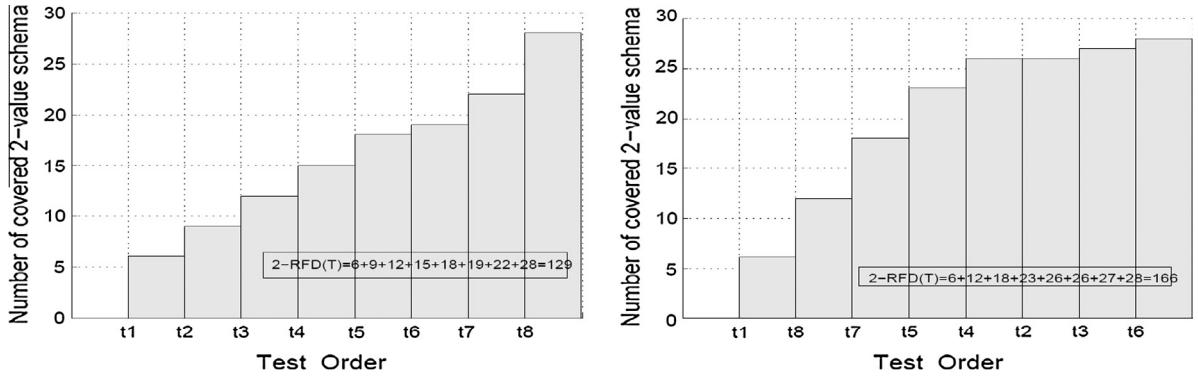

Fig. 1. Different test orders have different 2-RFD(T)s.

Consequently, $total\text{-}cov(T)=\frac{\sum_{\tau=1}^{n}|T(\tau\text{-}value)|}{\sum_{\tau=1}^{n}|SUT(\tau\text{-}value)|}$.

$\tau\text{-}cov(T)$ gives the ratio between all the covered τ-value combinations by T and all the τ-value combinations in *SUT*.

Definition 6 (τ-*way covering array*). Let $(c_{li})_{m\times n}, 1 \leqslant l \leqslant m, 1 \leqslant i \leqslant n$, be a $m \times n$ matrix, where each element c_{li} in column i is from the value set V_i of parameter c_i in the SUT. Each row is a test case. We denote row l as t_l for $1 \leqslant l \leqslant m$, and $t_l(i) = c_{li}$. If for every τ parameters $c_{i_1}, \ldots, c_{i_\tau}$, each value combination in $V_{i_1} \times \cdots \times V_{i_\tau}$ can be found in some row, the matrix $(c_{li})_{m\times n}, 1 \leqslant l \leqslant m, 1 \leqslant i \leqslant n$, is a τ-way covering array, denoted by $CA(m; \tau, a_1, a_2, \ldots, a_n)$. When $a_1 = a_2 = \cdots = a_n = a$, it can be denoted as $CA(m; \tau, a^n)$, otherwise it is a mixed τ-way covering array.

When $\tau\text{-}cov(T)$=100%, T is a τ-way covering array.

$total\text{-}cov(T)$ gives the ratio between all the covered value combinations by the test suite T and all the value combinations in *SUT*. $\tau\text{-}cov(T)$ and $total\text{-}cov(T)$ can measure the ITF detecting ability of a test suite T for SUT, without consideration of its size. Test suites with different sizes may have equal τ-*cov* and *total-cov*. To differentiate these, we define another metric, diversity.

SUT′ has 4 parameters and each of its parameters has 3 values. Then $|SUT(1\text{-}value)| = C_4^1 * 3 = 12, |SUT(2\text{-}value)| = C_4^2 * 3^2 = 54, |SUT(3\text{-}value)| = C_4^3 * 3^3 = 108$ and $|SUT(4\text{-}value)| = C_4^4 * 3^4 = 81$, in total $12 + 54 + 108 + 81 = 255$ value schemas. Suppose that CT uses a test suite (denoted as $T(CT)$) of 9 test cases to cover all 2-value schemas of *SUT*′. Assume this $T(CT)$ is a 2-way covering array $CA(9; 2, 3^4)$ as shown in Table 2, which can cover $9 * C_4^\tau$ τ-*value* schemas ($1 \leqslant \tau \leqslant 4$), that is, it covers 12 1-*value* schemas, 54 2-*value* schemas, 36 3-*value* schemas, and 9 4-*value* schemas.

Using Definition 5, the combination coverages of $T(CT)$ are 1-cov$(T(CT)) = \frac{12}{12} = 100\%$, 2-cov$(T(CT)) = \frac{54}{54} = 100\%$, 3-cov$(T(CT)) = \frac{36}{108} = 33\%$, 4-cov$(T(CT)) = \frac{9}{81} = 11\%$, and $total$-cov$(T(CT)) = \frac{12+54+36+9}{255} = 44\%$.

2.3. Diversity

To minimize the cost of detecting ITF, a test suite should be as small as possible to achieve high combination coverage. We can use diversity to quantify this property. Diversity measures the degree of distribution of test cases. Next we give its formal definition from [11].

Definition 7 (*Diversity*). Let $T = \{t_1, t_2, \ldots, t_m\}$ be a test suite for SUT; each test case $t_i, 1 \leqslant i \leqslant m$, can cover a distinct set $t_i(\tau\text{-}value)$ of C_n^τ τ-*value* schemas, that is $|t_i(\tau\text{-}value)| = C_n^\tau$. Let $T(\tau\text{-}value) = \bigcup_{i=1}^{m} t_i(\tau\text{-}value)$ be the set of all τ-*value* schemas covered by T. The τ-way diversity of T is $\tau\text{-}div(T) = \frac{|T(\tau\text{-}value)|}{m * C_n^\tau}$. Test suite T has m test cases, and can cover a total of $m(2^n - 1)$ value schemas, and $|\bigcup_{\tau=1}^{n} T(\tau\text{-}value)|$ is the number of all the value schemas T covered. The total diversity of T is $total\text{-}div(T) = \frac{|\bigcup_{\tau=1}^{n} T(\tau\text{-}value)|}{m*(2^n-1)}$.

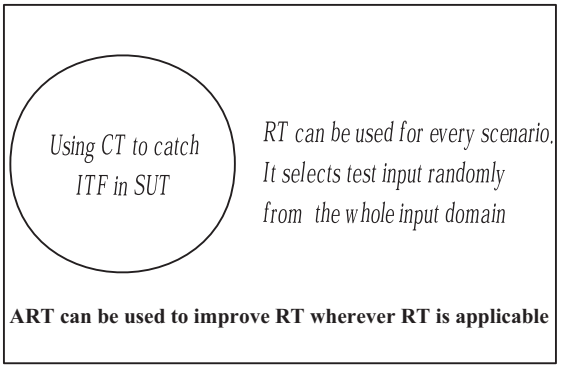

Fig. 2. The use scenario of CT, ART and RT.

From Definition 7, $m = |T|$, and when all schemas covered by each test case $t_i \in T$ are different $\tau\text{-}div = \frac{|T(\tau-value)|}{m*C_n^\tau} = \frac{\sum_{i=1}^{m}|t_i(\tau-value)|}{m*C_n^\tau} = \frac{m*C_n^\tau}{m*C_n^\tau} = 1$. The quality of T is the highest in τ-way diversity. As $\tau\text{-}div(T) = \frac{|T(\tau-value)|}{m*C_n^\tau}$, if a test suite T has a higher $\tau\text{-}div(T)$, it may imply that it uses fewer test cases (i.e. a smaller size $m = |T|$) and can cover more τ-way combinations ($|T(\tau\text{-}value)|$ is larger), and thus involves a relatively lower testing cost.

For the earlier example, 1-div($T(CT)$) = $\frac{12}{9*4} = 33\%$, 2-div($T(CT)$) = $\frac{54}{9*6} = 100\%$, 3-div($T(CT)$) = $\frac{36}{9*4} = 100\%$, 4-div($T(CT)$) = $\frac{9}{9*1} = 100\%$, and *total*-div($T(CT)$) = $\frac{12+54+36+9}{9*(2^4-1)} = 82\%$.

For an SUT with n parameters, each having a values, if $\tau\text{-}div(T) = 100\%$ and $\tau\text{-}cov(T) = 100\%$, the test suite T is an optimal τ-way covering array $CA(a^\tau; \tau, a^n)$. When $\tau\text{-}div(T) < 100\%$, another test suite may have higher diversity. But it is not always possible to get a^τ as the size of a τ-way covering array, in which case diversity cannot reach 100 percent (some τ-way combination must be repeated in the covering array).

2.4. Rate of Fault Detection (RFD)

Combination coverage measures the probability of a test suite detecting an ITF, while diversity measures its cost-effectiveness. It is also of interest to determine how many tests must be run for a fault to be found, if one is present. Two test suites of the same size may not necessarily find potential faults at the same rate.

As the number and distribution of faults are unknown, it is difficult to give a precise definition of the "rate of fault detection" as in [12,13]. But if we assume that every k-way interaction has the same probability of being faulty, and that faults arise independently, then we can give a formal definition of the rate of fault detection. Of course, both assumptions may not be realistic in practice, but in the absence of a specific fault model, these assumptions are reasonable.

Definition 8 (*RFD*). Let T={$t_1, t_2, \ldots, t_m$} be a test suite for SUT, with test cases in T to be run in the order: $t_1 - t_2 - \ldots - t_m$. Let T_i = {$t_1, t_2, \ldots, t_i$} for $1 \leqslant i \leqslant m$. If only τ-way interactions can be faulty, the rate of τ-way fault detection is $\tau\text{-}RFD(T) = \sum_{i=1}^{m}|T_i(\tau - value)|$. If all τ-way interactions ($1 \leqslant \tau \leqslant n$) can be faulty, the rate of fault detection is $total\text{-}RFD(T) = \sum_{i=1}^{m}|\bigcup_{\tau=1}^{n}T_i(\tau - value)|$.

According to [5], most software faults are triggered by no more than six factors interacting. Thus, we could suppose that all τ-way interactions for $1 \leqslant \tau \leqslant 6$ can be faulty, and define the rate of fault detection: $(1\text{–}6)\text{-}RFD(T) = \sum_{i=1}^{m}|\bigcup_{\tau=1}^{6}T_i(\tau\text{-}value)|$.

For example, $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$ in Table 3 is a test suite for *SUT*′. Different test orders $t_1 - t_2 - t_3 - t_4 - t_5 - t_6 - t_7 - t_8$ and $t_1 - t_8 - t_7 - t_5 - t_4 - t_2 - t_3 - t_6$ have different rates of fault detection, as shown in Fig. 1. From Definition 8, we can identify the relation between $total\text{-}RFD(T)$ and $\tau\text{-}RFD(T)$.

Theorem 9. *Let $T = \{t_1, t_2, \ldots, t_m\}$ be a test suite for SUT, with test cases in T run in the order: $t_1 - t_2 - \cdots - t_m$. Let $T_i = \{t_1, t_2, \ldots, t_i\}$ for $1 \leqslant i \leqslant m$. Then $total\text{-}RFD(T) = \sum_{\tau=1}^{n}\tau\text{-}RFD(T)$, and $(1\text{–}6)\text{-}RFD(T) = \sum_{\tau=1}^{6}\tau\text{-}RFD(T)$.*

Proof. We just need to prove that $total\text{-}RFD(T) = \sum_{\tau=1}^{n}\tau\text{-}RFD(T)$. By Definition 8, $total\text{-}RFD(T) = \sum_{i=1}^{m}|\bigcup_{\tau=1}^{n}T_i(\tau\text{-}value))|$, because for each T_i, $|\bigcup_{\tau=1}^{n}T_i(\tau\text{-}value))| = \sum_{\tau=1}^{n}|T_i(\tau\text{-}value)|$. It follows that

Table 4
$CA(11; 2, 3^4)$ by online CT, $T(onlineCT)$.

Index	c_1	c_2	c_3	c_4
1	A_2	B_1	C_0	D_1
2	A_0	B_0	C_0	D_0
3	A_2	B_2	C_2	D_2
4	A_1	B_2	C_1	D_1
5	A_1	B_1	C_2	D_0
6	A_0	B_1	C_1	D_2
7	A_0	B_0	C_2	D_1
8	A_2	B_0	C_1	D_0
9	A_1	B_2	C_0	D_2
10	A_1	B_0	C_1	D_2
11	A_0	B_2	C_0	D_0

$$total\text{-}RFD(T) = \sum_{i=1}^{m}\sum_{\tau=1}^{n}|T_i(\tau\text{-}value)| = \sum_{\tau=1}^{n}\sum_{i=1}^{m}|T_i(\tau\text{-}value)| = \sum_{\tau=1}^{n}\tau\text{-}RFD(T).$$

For $n = 6$, we have $(1\text{–}6)\text{-}RFD(T) = \sum_{\tau=1}^{6}\tau\text{-}RFD(T)$. □

2.5. Summary

When detecting ITF in SUT, we sample test cases like $t = (v_1, v_2, \ldots, v_n), t(i) = v_i$ and $v_i \in V_i, (1 \leqslant i \leqslant n)$ for the SUT. An ITF may be triggered by some k-MFS with $k > 0$ covered by these test cases. After we have tested the SUT with test suite T, the combination coverage of T allows us to estimate the probability that T hits an MFS, or the probability that some ITF remain in the SUT. The diversity of T permits us to measure the cost-effectiveness of testing. However, to quantify ability in detecting ITF faster, we need to consider the order of test cases.

Therefore when the goal is to find ITF in the SUT, we should select a test suite with high combination coverage, small size, high diversity, and a good testing order. In order to evaluate a testing method, we should determine whether it can deliver a high combination coverage test suite with high diversity and high RFD.

3. Using CT, RT and ART to hit ITF in SUT

CT, RT, and ART can each be used to detect ITF (hit MFS) in an SUT. To support a comprehensive comparison, for each we review its concepts, principles, procedures, examples, strengths, weaknesses, and related research areas in Sections 3.1, 3.2, and 3.3. In Section 3.4 we detail the stopping conditions used for CT, RT and ART.

3.1. Combinatorial Testing (CT)

3.1.1. Concept of CT

CT aims to detect failures triggered by an interaction among parameters of SUT, represented by a combination of some k parameter values, (i.e., a k-value schema) [3,1]. The basic function of CT is to detect MFS using a covering array as the test suite. Test suite generation for CT aims to construct a minimal covering array to cover the value combinations that may affect the software to the greatest degree. The typical use scenario of CT shown in Fig. 2 is discussed in Section 2.2 for the example *SUT*′. The most popular CT method is τ-way CT, denoted as CT(τ), which uses a τ-way covering array as test suite. A 2-way covering array $T(CT)$ with 9 test cases for *SUT*′ is shown in Table 2, which includes 9 test cases. CT covers all k-value schemas with $k \leqslant \tau$. This is illustrated in

Section 2.2 for SUT', where $CA(9;2,3^4)$ achieves 100% coverage for both 1-cov and 2-cov.

To examine the different degrees of interaction among parameters of an SUT efficiently and economically, the variable strength covering array has been proposed [14]. In a variable strength covering array, for some specific τ parameters, $c_{i_1}, c_{i_2}, \ldots, c_{i_\tau}$, where τ is variable rather than fixed, every value combination in $V_{i_1} \times \cdots \times V_{i_\tau}$ can be found in at least one row.

3.1.2. Principle of CT

Because many factors with mutual interactions may affect the SUT, it is desirable to employ a test suite covering all interactions. However, such an exhaustive test suite may be too large. Consequently there is a tradeoff between testing efficiency and cost. CT addresses this by covering only the interactions among few parameters, which is justified by empirical studies [5,15].

3.1.3. Method of CT

CT can be classified into offline and online methods. Offline CT(τ) generates a test suite (a τ-way covering array) first, then uses this test suite to test software. Online CT(τ) generates the next test case after the previous one has been run. There are many generation methods for τ-way covering arrays, including mathematical and computational ones. Many of them are deterministic, can deliver some suboptimal or optimal τ-way covering array, and are usually offline. In offline CT, the test suite can be chosen from the best covering arrays generated by any method. For small test suites, we can then compute a deterministic best test order by prioritizing the test suite using the RFD indicator.

To compare with RT and ART, we use an online greedy algorithm like AETG [16] to generate the "best" τ-way covering array. Algorithm 1 gives the procedure to generate one test case at a time to cover as many uncovered τ-value schemas as possible, and then test the SUT until the stopping condition is met. Stopping conditions are given in Section 3.4.

Algorithm 1. Combinatorial Testing (online CT(τ))

S *is the stopping condition* S_n *or* S_c; v_i is one of the values for the ith parameter c_i in SUT; $T = \emptyset$; // The set of the generated test cases;
Pw is the set of all uncovered τ-*value* schemas for SUT;
while ($\neg$**S**)
1. Select a factor c_i involved in the largest number of uncovered τ-*value* schemas. In the case of multiple candidate factors, choose c_i randomly.
2. Assign a value v_i to factor c_i.
3. Repeat steps 1–2 until all factors are assigned values, to get a test case.
4. Repeat steps 1–3 10 times to generate a candidate set with 10 test cases, then choose a test case t that hits the most uncovered τ-*value* schemas.
5. Test SUT with the test case t; $T = T \bigcup \{t\}$; $Pw = Pw - t(\tau\text{-}value)$;

End while;

3.1.4. Example of CT

We use 2-way coverage as an example, but in later experiments we use 2-way, 3-way and 4-way coverage. For the example, we use Algorithm 1 (online CT(τ)) to test SUT' to generate the 2-way covering array in Table 4 for SUT'. This online CT achieves 100% 2-way coverage with 11 test cases. The testing procedure ends when it uses the assigned number of test cases S_n, or achieves 100% 2-way coverage S_c.

3.1.5. Strengths and weaknesses of CT

CT uses a covering array as the test suite, which aims to test as many parameter value combinations as possible in order to detect failures triggered by parameter interactions. Some faults can be exposed by testing interactions among a small number of parameters, but not by testing individual parameter values. Hence CT can be very effective for certain types of applications, with performance approaching that of exhaustive testing while using fewer test cases [15,5]. Being specification-based, CT requires no knowledge about the implementation of SUT. Also, the specification required is "lightweight", because we only need to know the system settings to identify the input parameters and their possible values. Test generation can be automated, which is key to wide industrial acceptance.

Although CT is useful in detecting ITF, it can create a false confidence because: (1) It does not test all possible parameter combinations. (2) If parameters and their values are improperly selected, this reduces failure detection capability. (3) If we fail to identify interactions among parameters in SUT, CT does not test those "missed" interactions.

3.1.6. Related research on CT

CT research focuses on six areas [3]: (1) modeling to identify parameters and their values, interactions and constraints in SUT; (2) generating test suites, primarily focused on methods to generate a small test suite to cover as many parameter combinations as possible; (3) prioritization to search the best order of test execution to find faults earliest and most economically; (4) applications through empirical studies to explore the improvement of testing procedures; (5) debugging via failure characterization or fault diagnosis; and (6) measuring the combination coverage and the effectiveness of fault detection, and the degree to which CT contributes to the improvement of software quality.

3.2. Random Testing (RT)

3.2.1. Concept of RT

RT is a fundamental software testing method, which is easy to apply and has been extensively used. RT avoids complex analysis of program specifications or structures by selecting test cases randomly. It can be used in almost all the software testing scenarios shown in Fig. 2, especially in the absence of specifications and source code. It has been compared against CT [11,6].

Table 5
$CA(13;2,3^4)$ by ART, $T(ART)$.

Index	c_1	c_2	c_3	c_4
1	A_2	B_0	C_0	D_2
2	A_0	B_0	C_2	D_0
3	A_1	B_1	C_1	D_2
4	A_1	B_2	C_0	D_0
5	A_2	B_2	C_1	D_1
6	A_0	B_2	C_2	D_2
7	A_2	B_1	C_2	D_1
8	A_0	B_0	C_0	D_1
9	A_0	B_0	C_1	D_1
10	A_0	B_1	C_0	D_0
11	A_1	B_0	C_2	D_0
12	A_2	B_2	C_1	D_0
13	A_1	B_0	C_1	D_1

Algorithm 2. Random testing

S *is the stopping condition* S_n *or* S_c;
n is the number of parameters in SUT;
a_i is the number of values for the ith parameter c_i in SUT;
$v_{i[j]}, 0 \leqslant j < a_i$, is one of the values for the ith parameter c_i in SUT;
$T = \emptyset$; // The set of the generated test cases;
Pw is the set of all uncovered τ-*value* schemas for SUT;
while ($\neg$**S**)
    do {for $i = 1$ to n
        $t(i) = v_{i[\text{rand}(a_i)]}$; // [rand($a_i$)] generates integer j ($0 \leqslant j < a_i$) randomly
        End for;
        }*while*($t \in T$)
    Test SUT with t;
    $T = T \cup \{t\}$;
    $Pw = Pw - t(\tau\text{-}value)$;
End while;

3.2.2. Principle of RT

Because exhaustive testing is typically impossible, we must use limited testing resources to generate good test cases having high probability of detecting errors [17]. Tests generated randomly avoid potential bias on the part of the tester. RT is simple in concept and easy to implement, and its application does not require much knowledge of SUT. It has become a core part of other testing methods [2]. For example, in path coverage testing, we can first generate test cases randomly, then run them and check path coverage when it is difficult to generate test cases by solving path constraints [18].

3.2.3. Method of RT

The procedure of RT to hit the ITF for SUT is Algorithm 2. When a new test case t, $t = (v_1, v_2, \ldots, v_n)$, is generated at random, if t is different from the previously generated test cases, it is added to the test suite. These steps are repeated until the stopping condition is met.

3.2.4. Example of RT

We use RT to test SUT'. Table 6 shows that RT uses 18 test cases to cover all the 2-way combinations for SUT'. RT's test suite, $T(RT)$, is twice as big as $CA(9; 2, 3^4)$ in Table 2 and has 7 more test cases than $T(onlineCT)$ in Table 4. Its size is twice that of $T(CT)$ from Table 2.

3.2.5. Strengths and weaknesses of RT

The effectiveness of RT in detecting software failures is very controversial [19]. Because it uses almost no information about SUT, it has been considered a naive testing strategy and probably the poorest methodology [20]. When all inputs are treated uniformly, the effectiveness in detecting failures depends on the percentage of inputs associated with failures, while its effectiveness in covering elements depends on the percentage of inputs associated with elements to be covered. On the other hand, when inputs are treated differently according to the operational profile, RT's effectiveness in detecting failures also depends on the input weighting defined.

Because RT can test software without any bias, it has been shown to be useful [21], and is recommended as a final step of the testing activity [22]. RT has the following advantages: (1) low overhead, easy to implement and automate; (2) lightweight, requiring little information from the software specification and source code; (3) generates test cases that are less likely created by human testers, testing software without any bias so as to detect critical software failures that are not considered by human testers nor covered by deterministic approaches [23]; and (4) quantitative estimates of the software's operational reliability can be inferred from random testing [24].

3.2.6. Related research on RT

Research on RT has mainly concerned analysis and comparison on the effectiveness of RT, and its applications in real-life projects [24]. Since Myers claimed that "Probably the poorest testing methodology is random input testing" in the 70s, some researchers believe that RT cannot be as effective because many interesting tests have very little chance of being created at random [25]. Indeed, some empirical studies found that RT achieves less code coverage than systematic testing techniques, including chaining [26], model checking and symbolic execution [27].

Duran and Ntafos [21] defend RT, putting the problem on a well-defined formal base by comparing random testing to partition testing (dividing the input domain into nonoverlapping subdomains and selecting one test input from each), using simulations. Their results showed that RT may be more cost-effective. Hamlet and Taylor [28] considered these results counterintuitive. However, their simulation experiments essentially confirmed the observations by Duran and Ntafos. Weyuker and Jeng compared the two testing approaches from an analytical point of view, with results pointing in the same direction [29]. But Gutjahr proved that partition testing should be better than RT [30] by generalizing the result from Weyuker and Jeng. Andrea et al. conducted a further study, in which RT was more effective and predictable than previously thought [31].

Despite many debates, RT is widely used, for example, in testing the robustness of Windows NT applications, Mac OS X applications, object-oriented software, Haskell programs, Java JIT Compiler, embedded software and software model checking [24].

3.3. Adaptive Random Testing (ART)

3.3.1. Concept of ART

ART aims to randomly select test cases, but also to spread them evenly. The intuition is that new test cases that are located away from the test cases that reveal no failure are more likely to reveal failures. Whenever RT can be used, ART can also be used (Fig. 2). But ART has not been used to test SUT in the same manner as CT.

3.3.2. Principle of ART

ART was proposed by Chan et al. [32] to improve the fault detection capability of RT. ART exploits information on executed test cases, particularly successful test cases, to guide the random

Table 6
$CA(18; 2, 3^4)$ by RT, $T(RT)$.

Index	c_1	c_2	c_3	c_4
1	A_0	B_0	C_1	D_1
2	A_1	B_0	C_2	D_2
3	A_1	B_1	C_0	D_1
4	A_1	B_0	C_2	D_1
5	A_0	B_0	C_1	D_2
6	A_1	B_2	C_1	D_1
7	A_1	B_2	C_2	D_1
8	A_1	B_2	C_2	D_0
9	A_2	B_0	C_1	D_1
10	A_2	B_0	C_2	D_2
11	A_0	B_2	C_0	D_2
12	A_2	B_1	C_1	D_2
13	A_2	B_1	C_0	D_1
14	A_1	B_0	C_0	D_0
15	A_0	B_1	C_2	D_1
16	A_0	B_2	C_1	D_0
17	A_2	B_2	C_1	D_0
18	A_1	B_1	C_1	D_0

test case generation. Because failure-causing inputs are often clustered together to form one or more contiguous regions, subsequent test cases that are close (similar) to successful test cases are less likely to hit the failure-causing region than those that are far away. Hence, ART promotes diversity among random test cases [2]. The distance metric (or dissimilarity metric) is a key component in ART used to measure the distance (difference) among inputs.

3.3.3. Method of ART

We use ART with fixed size candidate set (FSCS ART) to hit the ITF in Algorithm 3. Each time q test cases are randomly generated to form a candidate set. For each candidate test case, we compute its distance to each of the previously generated d test cases. A candidate test case with the largest distance is chosen. The procedure terminates when the stopping condition is met.

The performance of ART depends on the distance measure used. Because the test case space is typically numeric or can be transformed into numeric, the Hamming distance can be used.

Definition 10 (*Hamming Distance*). The Hamming distance, $Ham(t_1, t_2)$, between two test cases t_1 and t_2 is the number of parameters for which they have different values. When $T = \{t_1, t_2, \ldots, t_k\}$ is a set of test cases and t_i' is a test case, the Hamming distance between t_i' and T is: $Ham(t_i', T) = \min_{t_j \in T} Ham(t_i', t_j)$.

In each iteration, a candidate test case t_i' with the largest Hamming distance to T is chosen, that is, $Ham(t_i', T) = \max_{t_j' \in Cand(m)} Ham(t_j', T)$. Hamming Distance is not developed to target ITF specifically, but may deliver good diversity. To improve effectiveness, schema distance is developed here. Our goal is to generate test cases to hit the MFS, which triggers ITF. If a generated test case t failed to reveal any failure, all the value schemas covered by t do not trigger any ITF. When we select the next test case t', the best candidate should have the most different value schemas from t. The best test case is one that covers the largest number of uncovered value schemas.

Definition 11 (*Schema Distance*). Let $P(t, \tau)$ denote the set of all τ-*value* schemas covered by test case t. The schema distance between test cases t_1 and t_2 is: $Schema(t_1, t_2) = |P(t_1, \tau) - P(t_2, \tau)|$. The schema distance between a test case t_i' and a test set T is: $Schema(t_i', T) = |P(t_i', \tau) - \bigcup_{t_j \in T} P(t_j, \tau)|$.

Algorithm 3. Adaptive Random Testing (FSCS ART)

```
S is the stopping condition either S_n or S_c;
q is the fixed size of candidate set;
Pw is the set of all uncovered τ − value schemas for SUT;
T = ∅; //The set of the generated test cases;
while (¬S)
    bestd = 0; //The largest distance of a candidate test case t_j
  to T;
    bestt: the best test case candidate;
    for j = 1 to q//Distance calculation.
        Randomly generate a test case t_j;
        fitness(t_j) = distance(t_j, T);
        if (bestd < fitness(t_j))
            bestt = t_j and bestd = fitness(t_j);
    if bestd ≠ 0
        Test SUT with bestt;
        T = {bestt} ∪ T;
        Pw = Pw − bestt(τ-value);
End while;
```

Table 7
Quality metrics of CT, RT and ART with stopping condition S_n.

Stopping condition S_n	Offline CT	Online CT	RT	ART
2-*cov*	100%	93%	59%	85%
2-*div*	100%	93%	59%	85%
2-*RFD*	270	263	248	262

3.3.4. Example of ART

Table 5 shows the 2-way covering array, $T(ART)$, for SUT' generated by ART, which is smaller than the 2-way covering array $T(RT)$ in Table 6. However, ART requires more computation time.

3.3.5. Strengths and weaknesses of ART

The development of ART is inspired by the observed clustering of failure-causing inputs. By diversifying test cases, failure-causing inputs can be found more quickly. ART incorporates the advantages of random testing, and the advantage of test case diversity.

Empirical studies of real-life programs have shown that ART can significantly enhance RT in terms of fewer test cases needed to detect the first failure (F-Measure) and in terms of fewer test cases required to achieve a fixed degree of code coverage. But Arcuri et al. studied one algorithm, FSCS-ART, and found that it does not work as well as expected in practice [19]. They gave two reasons: (1) the calculation of distances among inputs overshadows the cost reduction that FSCS-ART yields in terms of number of executed test cases; and (2) diversity in test cases is intuitive, but FSCS-ART may not be an appropriate way to achieve it. Of course, FSCS-ART is only one of many ART algorithms. The cost of generating test cases depends on the specific algorithm used, and a cheaper ART algorithm can be considered.

3.3.6. Related research on ART

Since ART was proposed by Chen et al. [32], many research papers have appeared. Improvements on the basic ART have been classified into five directions [24]: (1) avoiding redundant and illegal inputs, and finding useful inputs. (2) combining random and systematic approaches. (3) coverage-oriented approaches. (4) spreading the input values evenly, including Mirror ART, ART by restriction, ART by localization, ART by partitioning, ART by lattice, ART by failure information and test profile, ART by balancing, ART by Voronoi diagram and ART by distribution metric. (5) generalizations such as Fuzzy ART [33].

ART has been used in testing Object-Oriented software, aspect-oriented software and embedded software [24]; for test case prioritization [34], model based test case selection [35], and enlarged and high dimensional input domain [36]. Limitations of ART have been examined [37], including studying a lower bound on the F-measure achievable by any test method; ART's F-measure is close to the lower bound [38].

3.4. Stopping conditions

Stopping condition S_n indicates that testing stops when the number of generated test cases reaches an assigned number. We compare the combination coverage, diversity and RFD achieved by test suites of the same size as the τ-way covering array from offline CT. For example, $T(CT)$ in Table 2 is the covering array for offline CT, with a size of 9. (Henceforth we use $T(offlineCT)$ to denote $T(CT)$.) Then we use online CT (Algorithm 1) to generate the first 9 test cases in Table 4, RT (Algorithm 2) to generate the first 9 test cases in Table 6, and ART (Algorithm 3) to the first 9 test cases in Table 5. Table 7 reports results for the quality metrics when using 9 test cases. Here, offlineCT has the highest 2-*RFD*, 2-cov, and 2-div.

Table 8
12 instances of SUT $CA^i(N; \tau, a_1 \times a_2 \cdots \times a_n)(1 \leqslant i \leqslant 12, \tau = 2, 3, 4)$.

CA^1:CA(190; 2, 10^{30})	CA^2:CA(10; 2, 2^{100})	CA^3:CA(155; 2, 10^{20})	CA^4:CA(71; 2, $7^6 6^7 5^6$)
CA^5:CA(15; 2, 3^{13})	CA^6:CA(120; 2, 8^{40})	CA^7:CA(250; 2, $15^9 10^6$)	CA^8:CA(24; 2, 4^{10})
CA^9:CA(108; 2, 8^{20})	CA^{10}:CA(48; 2, 6^{10})	CA^{11}:CA(28; 2, 4^{20})	CA^{12}:CA(65; 2, 6^{20})
CA^1:CA(2584; 3, 10^{30})	CA^2:CA(33; 3, 2^{100})	CA^3:CA(2259; 3, 10^{20})	CA^4:CA(670; 3, $7^6 6^7 5^6$)
CA^5:CA(50; 3, 3^{13})	CA^6:CA(1016; 3, 8^{40})	CA^7:CA(4040; 3, $15^9 10^6$)	CA^8:CA(112; 3, 4^{10})
CA^9:CA(1016; 3, 8^{20})	CA^{10}:CA(393; 3, 6^{10})	CA^{11}:CA(152; 3, 4^{20})	CA^{12}:CA(624; 3, 6^{20})
CA^1:CA(47410; 4, 10^{30})	CA^2:CA(98; 4, 2^{100})	CA^3:CA(29269; 4, 10^{20})	CA^4:CA(6157; 4, $7^6 6^7 5^6$)
CA^5:CA(219; 4, 3^{13})	CA^6:CA(18864; 4, 8^{40})	CA^7:CA(65211; 4, $15^9 10^6$)	CA^8:CA(508; 4, 4^{10})
CA^9:CA(8184; 4, 8^{20})	CA^{10}:CA(3210; 4, 6^{10})	CA^{11}:CA(760; 4, 4^{20})	CA^{12}:CA(5973; 4, 6^{20})

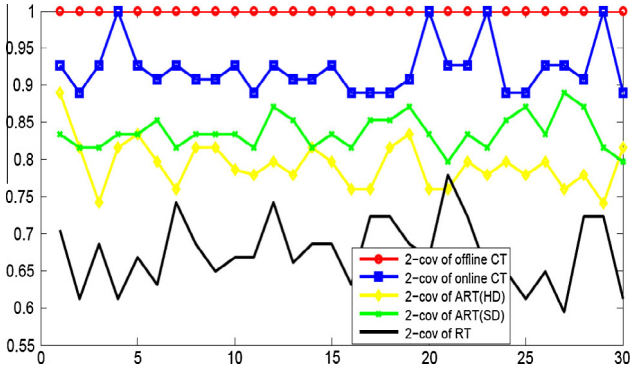

Fig. 3. 2-cov of the same size test suite.

Stopping condition S_c indicates that testing stops when a specified percentage c of combination coverage is reached; usually we take $c = 100\%$. When we require 100% τ-way coverage, CT, RT and ART generate a series of test cases to cover all τ-*value* schemas. They may generate different numbers of test cases in the process. CT needs the fewest test cases, ART needs more, and RT needs the most. For example, offlineCT uses $T(CT)$ in Table 2; onlineCT generates 11 test cases in Table 4; RT generates 18 test cases ($T(RT)$, Table 6); and ART generates 13 test cases ($T(ART)$, Table 5).

4. Comparing RT, ART and CT in hitting MFS

4.1. Experimental design

Now we describe the experimental design and implementation. We have implemented Algorithms 1–3, to generate $\tau - way$ covering arrays for $\tau = 2, 3, 4$. We implement two versions for ART, ART(HD) using Hamming Distance and ART(SD)(τ) using Schema Distance. For offline CT(τ), we use the smallest known τ-*way* covering arrays, shown in Table 8. (References for each of the constructions of Table 8 are given in an appendix.) We developed an analysis tool for test suites that measures the combination coverage (τ-*cov*), diversity (τ-*div*) and RFD (τ-*RFD*) for $\tau = 2, 3, 4$.

To be representative, we select $12 \times 3 = 36$ instances of SUT in Table 8, including small scale (such as 3^{13}), middle scale (such as 8^{20}) and large scale covering arrays (such as 2^{100}), and some mixed covering arrays (such as $7^6 6^7 5^6$), for $\tau = 2, 3, 4$. For each instance, we repeat experiments 30 times to evaluate online CT(τ), offline CT(τ), RT, ART(HD), and ART(SD)(τ) with the analysis tool.

4.1.1. Fixed test suite size (S_n)

We generate online CT's, RT's, and ART's test suites, having the same size as that of offline CT. Then the analysis tool computes RFD, combination coverage and diversity. Each method involves some randomness, and hence we consider the averages over 30 trials.

Result are shown for our running example SUT' in Figs. 3 and 4 (Because 2-cov and 2-div produce the same result in this case, we just present 2-cov). We can conclude that (1) offline CT is the best; (2) online CT is better than ART(HD) and ART(SD); (3) ART(SD) is generally better than ART(HD); and (4) ART(HD) and ART(SD) always perform better than RT.

4.1.2. Fixed combination coverage (S_c)

In the S_c experiment, we compare the sizes of the test suite needed by online CT, RT, ART(HD) and ART(SD) to the size of τ-way covering array for offline CT. We compare average sizes over 30 trials. Results are shown for our running example SUT' in Fig. 5. Here ART(HD) and ART(SD) are always better than RT, and CT is always better than ART. The average size of an RT test suite is 30, while that of ART (HD), ART (SD) and online CT are 19.7, 14.2, and 10.8 respectively.

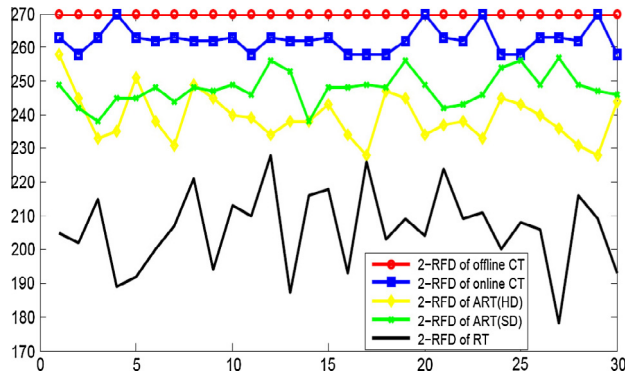

Fig. 4. 2-RFD of the same size test suite.

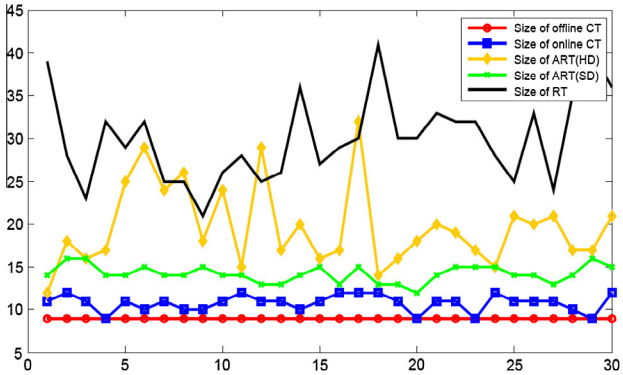

Fig. 5. Test cases needed for fixed coverage.

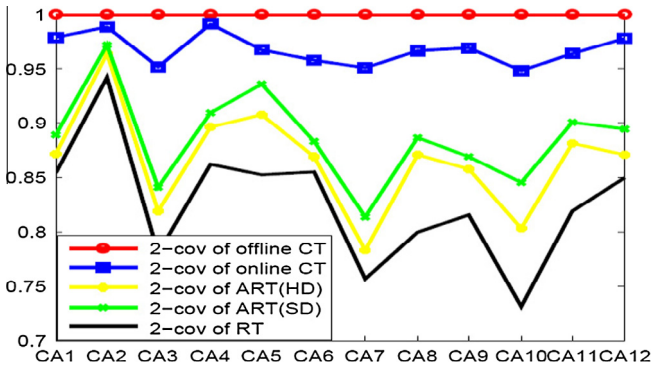

Fig. 6. $S_n(2)$ (2-cov).

4.2. Threats to validity

Before presenting experimental results, we state concerns that may impact validity.

1. We treat only 36 instances of SUT. Different results may be obtained for more or different instances.
2. Online CT, ART(HD), ART(SD) and RT are random methods, and we just observe 30 runs and use the average. More runs may alter results somewhat.
3. Smallest possible covering arrays are not determined in general, and instead we use the smallest *known* in Table 8. If smaller covering arrays exist, that can affect our result.
4. ART is affected by the size of candidate set and distance measure used. We use 10 candidates for ART(HD) and ART(SD). Different sizes could affect the results.

4.3. Results

For each instance $CA^i(\mathrm{N};\tau, a_1 \times a_2 \cdots \times a_n)$ with $1 \leqslant i \leqslant 12$, $\tau = 2,3,4$ from Table 8, we run experiments on S_n and S_c. Next we present the results.

4.3.1. Result of S_n experiments

We first consider $CA^i(\mathrm{N};\ \tau, a_1 \times a_2 \cdots \times a_n)$ for $1 \leqslant i \leqslant 12$, with $\tau = 2$, the first three rows of Table 8. (This group of experiments is denoted as $S_n(2)$). We use offline CT to assign the fixed size to be tested. Results for the S_n experiments are shown in Figs. 6–14.

The 2-cov of the same sized test suite generated by online CT($\tau = 2$), ART(HD), ART(SD)($\tau = 2$) and RT are very different, and satisfy: 2-cov($offlineCT$) > 2-cov($onlineCT$) > 2-cov($ART(SD)$ ($\tau = 2$)) > 2-cov($ART(HD)$) > 2-cov(RT) (Fig. 6). The differences of these test suites in 3-cov are much smaller (Fig. 7), and the differences in 4-cov are negligible (Fig. 8).

All 12 2-way covering arrays from offline CT have the highest 2-div. Obvious differences among the test suites generated by the other four methods suggest: 2-div($offlineCT$) > 2-div($onlineCT$) > 2-div($ART(SD)(\tau = 2)$) > 2-div($ART(HD)$) > 2-div(RT) (Fig. 9). However, their 3-div (Fig. 10) and 4-div (Fig. 11) are increasing, and differences are becoming smaller as τ increases.

To compare the relative RFD, we compute $\frac{RFD(method)}{RFD(offlineCT)}$ to produce Figs. 12–14. In these comparisons, $0.8 \leqslant \frac{2\text{-}RFD(method)}{2\text{-}RFD(offlineCT)} \leqslant 1.05$, $0.91 \leqslant \frac{3\text{-}RFD(method)}{3\text{-}RFD(offlineCT)} \leqslant 1.03$, $0.94 \leqslant \frac{4\text{-}RFD(method)}{4\text{-}RFD(offlineCT)} \leqslant 1.01$. Hence the methods have closer τ-RFD when τ increases, and so they have similar failure detecting abilities for 3-MFS and 4-MFS. We can conclude that pairwise testing does not differ significantly from the same sized RT and ART in detecting 3-MFS and 4-MFS. But there is a big difference in detecting 2-MFS.

Similar results arise in the $S_n(3)$ experiments, using $\tau = 3$ instead and the second three rows in Table 8. The results are shown in Figs. 15–23. Except a few obvious differences in 3-cov, 3-div and 3-RFD among the fixed size test suites, these differences become smaller when $\tau = 2$ and 4.

For the $S_n(4)$ experiments, with $\tau = 4$ and the third three rows in Table 8, results are shown in Figs. 24–32. Except a few obvious differences in 4-cov, 4-div and 4-RFD among the fixed size test suites, these differences become smaller and smaller when $\tau = 3$ and 2.

From all the $S_n(\tau)$ ($\tau = 2,3,4$) experiments, we conclude that τ- way CT makes an obvious improvement over the other methods in detecting k-$value$ MFS where $k = \tau$, but when $k > \tau$ or $k < \tau$, their differences are less pronounced.

4.3.2. Result of S_c experiments

In this experiment, different methods produce different sizes of test suite. For the sake of comparison, we record the rate $\frac{size(method)}{size(offlineCT)}$ for each instance, where $size(method)$ denotes the average number (over 30 runs) of test cases needed to reach τ-way coverage.

To reach 100% τ-way coverage (τ = 2, 3, 4), Figs. 33–35 show that for the most part:

$$1 \leqslant \frac{size(onlineCT)}{size(offlineCT)} \leqslant 2$$

$$2 \leqslant \frac{size(ART(HD))}{size(offlineCT)} \leqslant 3$$

$$3 \leqslant \frac{size(ART(SD))}{size(offlineCT)} \leqslant 7$$

$$4 \leqslant \frac{size(RT)}{size(offlineCT)} \leqslant 9$$

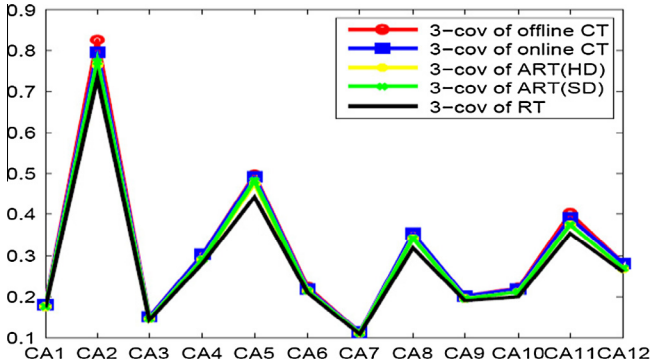

Fig. 7. $S_n(2)$ (3-cov).

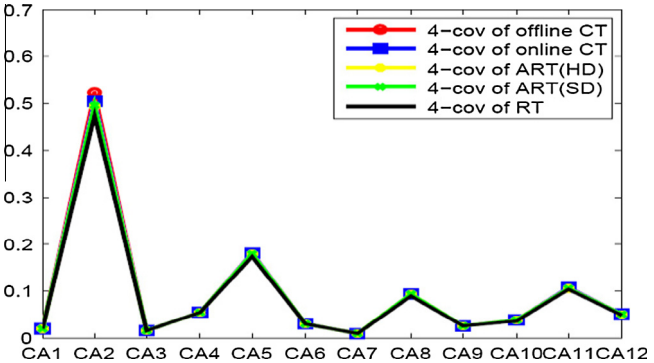

Fig. 8. $S_n(2)$ (4-cov).

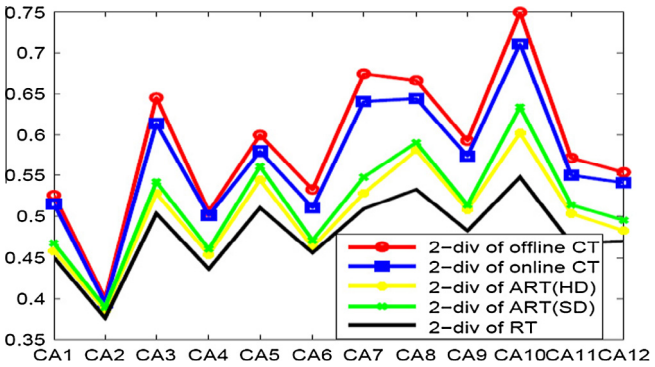

Fig. 9. $S_n(2)$ (2-*div*).

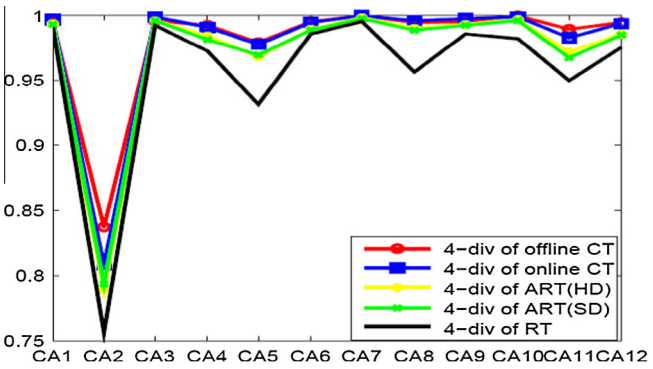

Fig. 11. $S_n(2)$ (4-*div*).

Each method makes test suites that are much larger than offlineCT, but onlineCT comes the closest.

4.4. Summary

Now we can answer our research questions:

RT is definitely an effective test method in detecting ITF. The average test suite size to achieve the fixed coverage is often more than four times that of the optimal size. Different runs of RT have slightly different performance. These results remain true as the number of parameters varies from 10 to 100, and the number of values varies from 2 to 15.

CT (both offlineCT and onlineCT) employ much smaller sized test suites to achieve the same coverage as RT. Their test suites achieves a higher combination coverage, diversity and RFD than those of RT. In τ-way CT, if the failure causing schema is *t-MFS* and $t \neq \tau$, there appears to be no significant difference between CT and RT in detecting *t-MFS*. CT can be improved by using a smaller covering array, as shown by the fact that offlineCT is normally better than onlineCT.

ART (ART(HD) and ART(SD)) definitely improve on RT with respect to combination coverage, diversity, RFD and test suite size needed for achieving τ-way coverage. This conclusion is supported by both the S_n and S_c experiments. ART can be improved by using better heuristics, such as a better distance measure and a larger candidate set, as shown by the improvement of ART(SD) on ART(HD) for detecting ITF.

While ART(HD) and ART(SD) improve upon RT, they never outperform CT. There is a substantial difference between the performance of RT and CT. For the metrics chosen and the experiments conducted, CT is better than ART (ART(HD) and ART(SD)) and ART is better than RT.

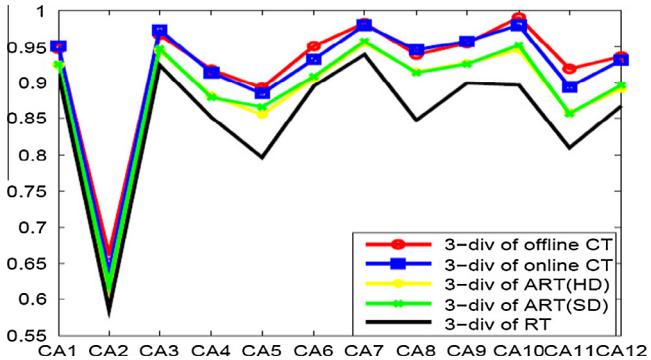

Fig. 10. $S_n(2)$ (3-*div*).

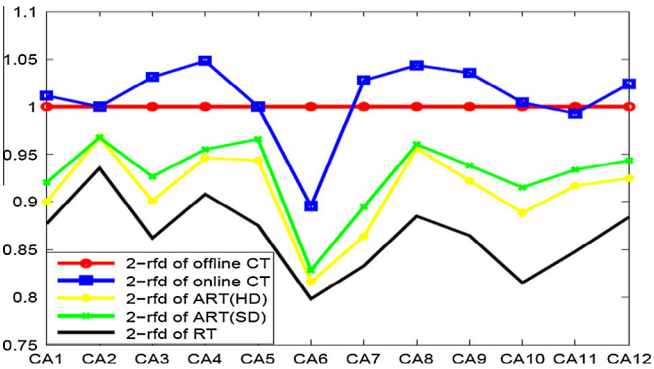

Fig. 12. $S_n(2)$ (2-*RFD*).

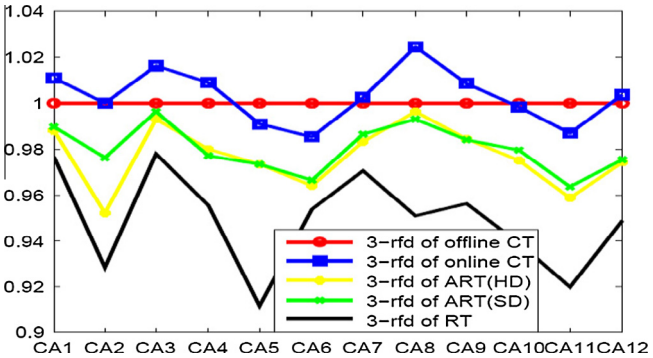

Fig. 13. $S_n(2)$ (3-*RFD*).

5. Comparisons with published studies

Many comparisons of CT and RT, and of RT and ART, have been made, but we are unaware of any study to compare CT to ART systematically. Such comparisons as exist focus on some specific aspect and may not be relevant to catching ITF. Our study is a first attempt to investigate the relationships among CT, RT and ART.

5.1. Comparisons between RT and CT

Mandl [39] used orthogonal Latin Squares, a special 2-way covering array, in the validation of an Ada compiler. He concluded that orthogonal Latin squares are a very efficient compromise between testing effort and information obtained, and pointed to RT as being "much less effective in detecting malfunctions and yields much

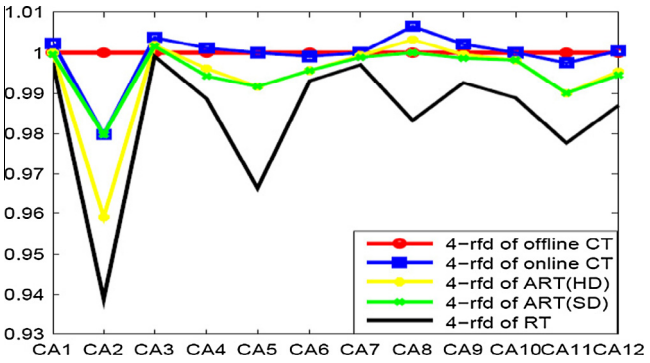

Fig. 14. $S_n(2)$ (4-*RFD*).

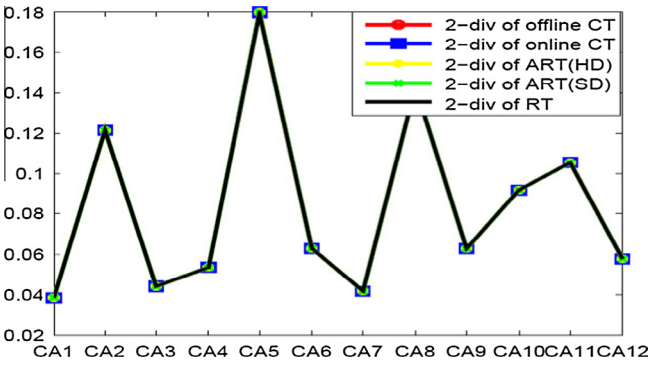

Fig. 18. $S_n(3)$ (2-*div*).

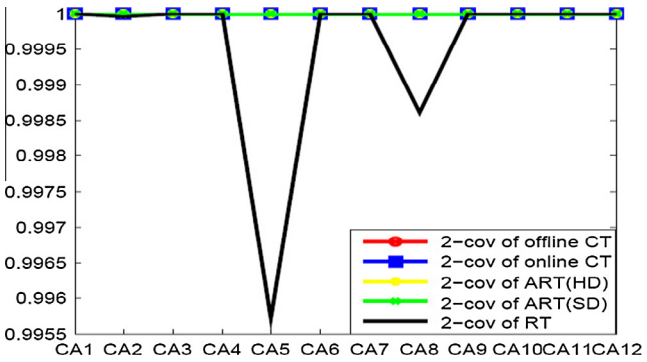

Fig. 15. $S_n(3)$ (2-*cov*).

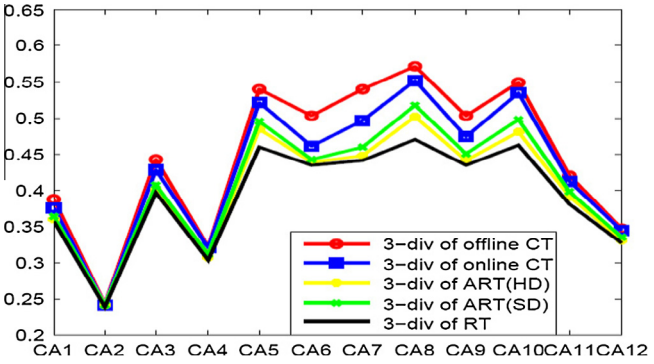

Fig. 19. $S_n(3)$ (3-*div*).

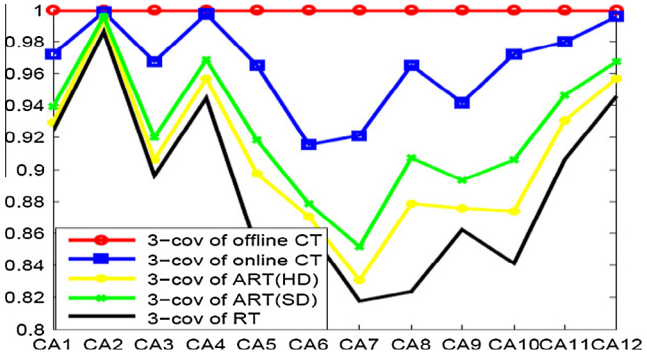

Fig. 16. $S_n(3)$ (3-*cov*).

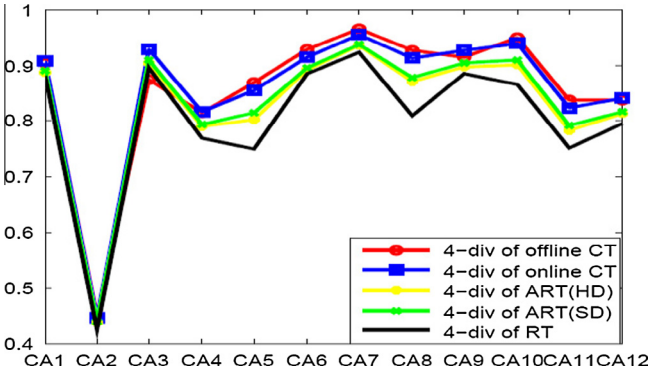

Fig. 20. $S_n(3)$ (4-*div*).

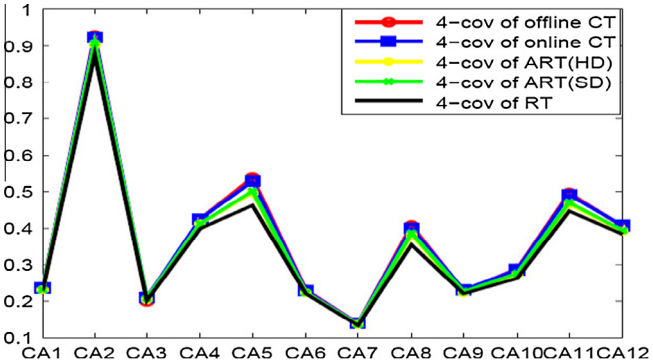

Fig. 17. $S_n(3)$ (4-*cov*).

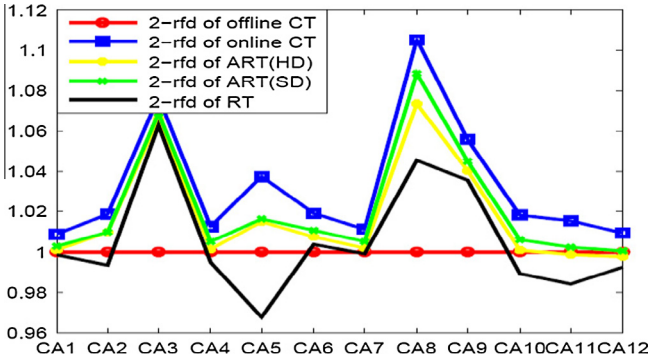

Fig. 21. $S_n(3)$ (2-*RFD*).

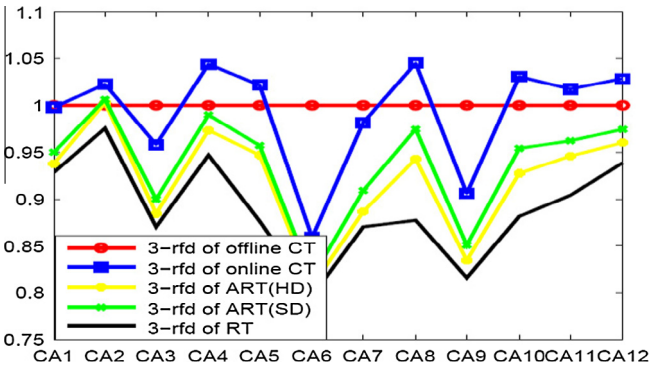

Fig. 22. $S_n(3)$ (3-*RFD*).

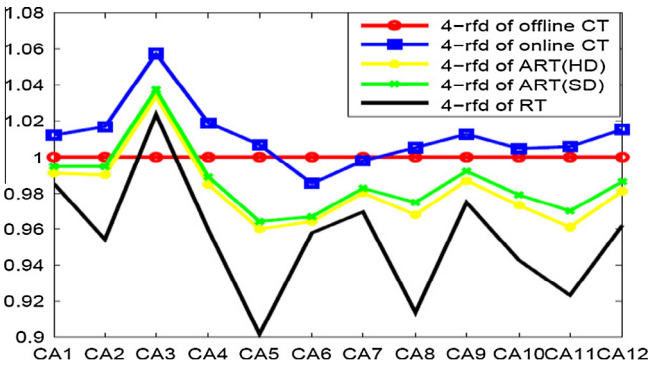

Fig. 23. $S_n(3)$ (4-*RFD*).

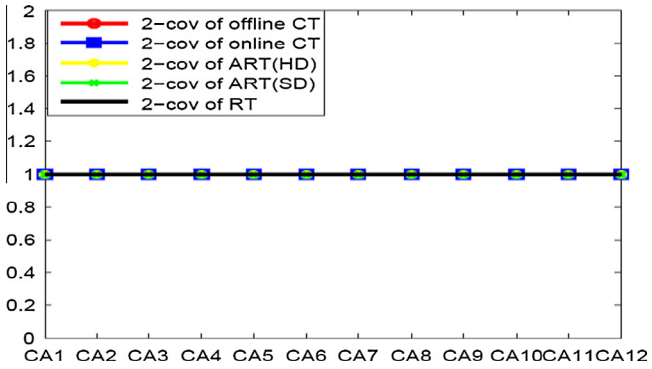

Fig. 24. $S_n(4)$ (2-*cov*).

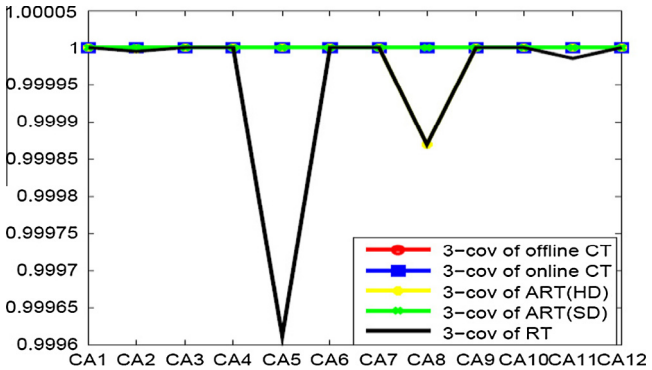

Fig. 25. $S_n(4)$ (3-*cov*).

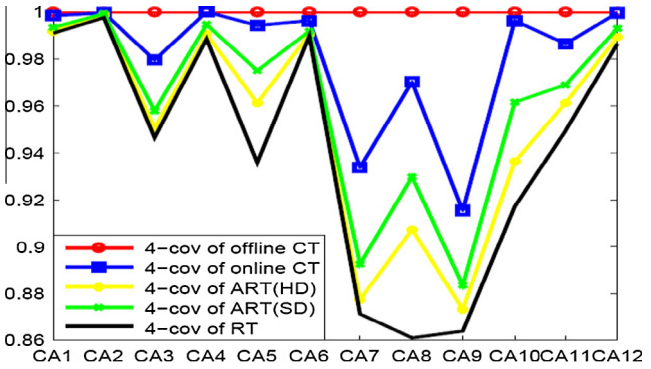

Fig. 26. $S_n(4)$ (4-*cov*).

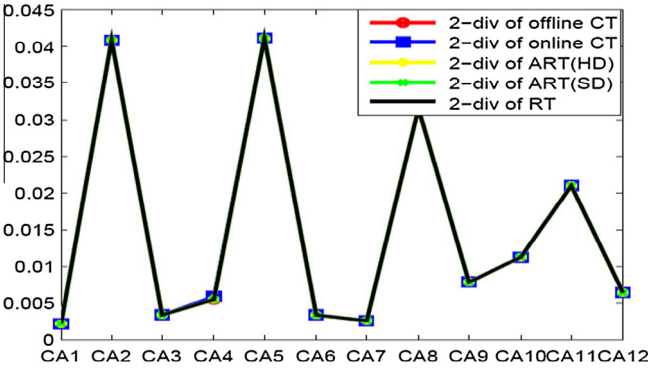

Fig. 27. $S_n(4)$ (2-*div*).

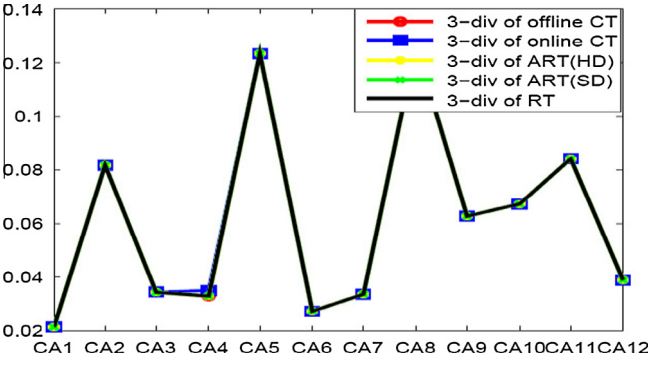

Fig. 28. $S_n(4)$ (3-*div*).

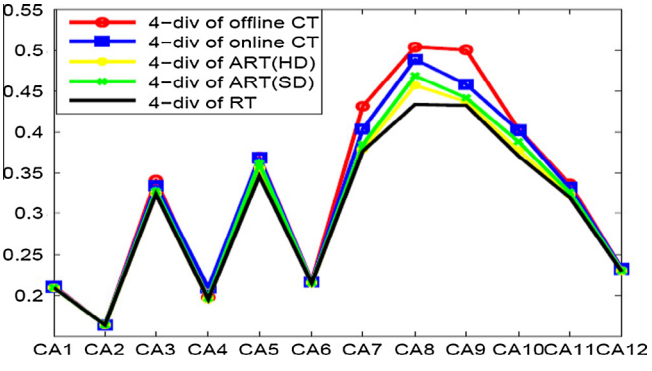

Fig. 29. $S_n(4)$ (4-*div*).

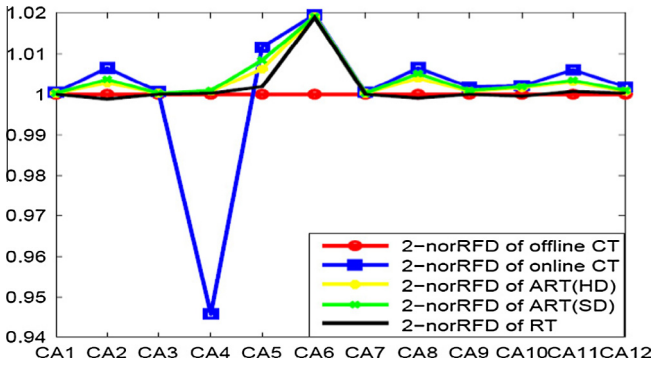

Fig. 30. $S_n(4)$ (2-*RFD*).

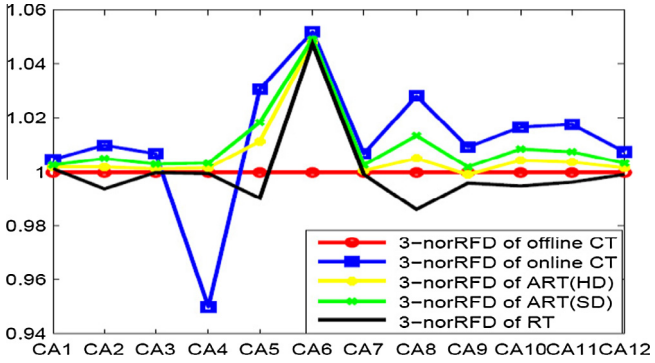

Fig. 31. $S_n(4)$ (3-*RFD*).

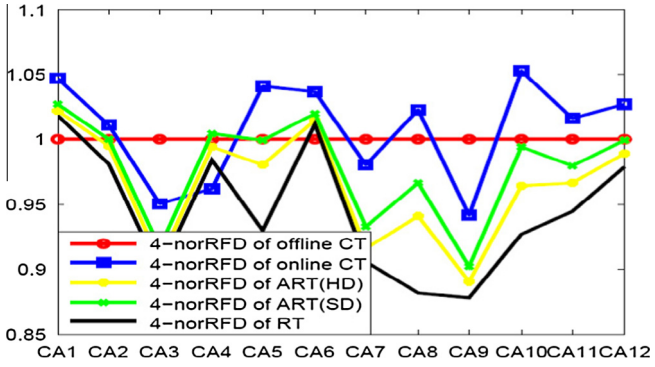

Fig. 32. $S_n(4)$ (4-*RFD*).

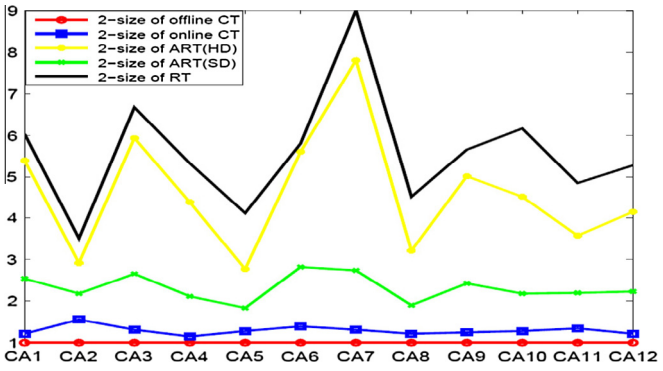

Fig. 33. Size comparison ($\tau = 2$).

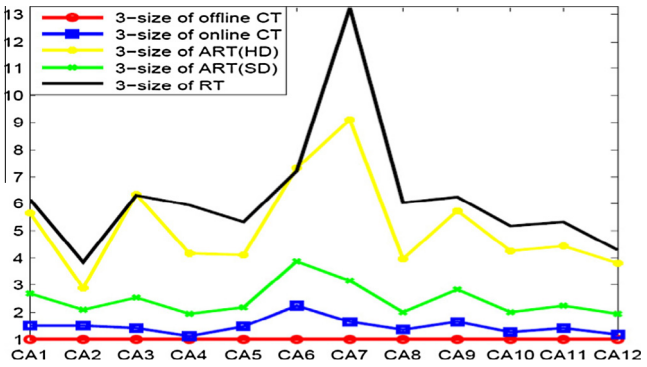

Fig. 34. Size comparison ($\tau = 3$).

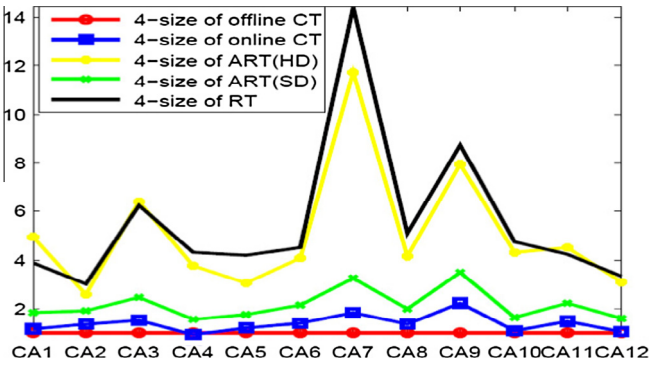

Fig. 35. Size comparison ($\tau = 4$).

less information when it does in fact detect something". This is the earliest intuition on the effectiveness of CT and RT, but lacks supporting data.

Dalal and Mallows [11] later observed that CT was almost the same as random testing in failure detecting effectiveness. They analyzed same-sized random test data sets and found that they covered a very high percentage of input pairs, as high as 90% in many cases. Based on this, one may not expect the failure detecting effectiveness of pairwise combination coverage testing and same-sized random test data sets to vary to a great degree [11]. The results from our experiments also confirm this observation about the coverage of RT. For example, for the 12 2-way covering arrays in Table 8, the same sized test suite generated by RT can cover 75–95% of all the 2-way combinations (see Fig. 6). For the 12 3-way covering arrays in Table 8, the same sized test suite generated by RT covers 80–98% of all the 3-way combinations (see Fig. 16).

A controlled study to compare the fault detection effectiveness of τ-way CT and RT [10] found no significant difference between them. The numbers of τ-way value combinations covered are very similar for τ-way CT and the same-sized suite for RT, especially when $\tau = 2$. But this study raises some concerns. First, all the failures are caused by injected mutants, and may have very few ITF. Secondly, the selected application may not fit CT,in that the parameters and their value combinations may have little interaction. Thirdly, the algorithm used for CT was not state-of-the-art.

Michael et al. compared the effectiveness of automatically generated tests constructed using random and t-way combinatorial techniques on safety related industrial code using mutation adequacy criteria. Their study showed that 2-way testing was not adequate as measured by mutants kill rate compared to hand generated test set of similar size, but that higher *t*-way CT could perform at least as well [7].

Kuhn et al. examined the fault detecting ability of RT and CT [40], and found 2-way CT detected slightly fewer deadlocks than an equal number of random tests, but 4-way CT produced a better result. They explained that was because most of ITF were triggered by 4-value schema. In their simulation study, they found that the test suite for RT was normally several times larger than the covering array used by CT to reach the same τ-way coverage. From our experiments, in detecting 4-MFS, 2-way CT has almost same ability as the same size test suite of RT (see Fig. 14), but 4-way CT is definitely better than the same size test suite of RT (see Fig. 32). The test suite needed by RT to achieve τ-way coverage is normally 4–9 times that of CT (see Figs. 33 and 34).

Arcuri and Briand made a formal analysis on the probability of ITF detection of RT and CT, and concluded that random testing became even more effective as the number of features increased and converged towards equal effectiveness with combinatorial testing. Given that combinatorial testing entails significant computational overhead in the presence of hundreds or thousands of features, the results could suggest that there are realistic scenarios in which random testing can outperform combinatorial testing in large systems [8].

Other studies indicate that CT is more effective than RT. For example, Dunietz et al. compared the code coverage of random designs without replacement vs. the coverage obtained from systematic designs (i.e. t-way adequate test sets) with the same number of test cases. They conclude that for block coverage, low factor t-way designs could be effective [41]. Nair et al. investigated random testing without replacement and no partitioning vs. partition based testing, and showed that in general partition testing should be more effective. The particular case of partition testing was an application of experimental design (t-way) and it showed that the probability of detecting the failure for simple random testing was significantly lower than partition based techniques [42]. Kobayashi et al. examined the fault detecting ability of specification based, random, anti-random and t-way techniques applied to testing logic predicates against mutations of those predicates. They conclude that 4-way tests were nearly as effective as specification techniques and better than both random and anti-random [43].

5.2. Comparisons between RT and ART

Many studies have compared ART with RT. Almost all show that ART can be better [9,2,31]. Although the comparison of failure detection between RT and ART has been undertaken, this comparison has not considered the case of hitting MFS in testing for ITF.

6. Conclusions and future work

RT is an effective and widely used method, ART was proposed to enhance RT, and CT has been specifically developed to test the ITF (hit the MFS). Although they have different objectives, there have been many debates about their effectiveness and usage. Our results support two strong conclusions. First, in hitting τ'-MFS with $\tau' \leqslant \tau$, τ-*way* CT is definitely better than the other testing methods when using the same size test suite as the size of the smallest covering array. Secondly, τ-*way* CT may not be better than the others in hitting τ'-MFS when $\tau' > \tau$.

Experiments support these conclusions. As in Section 4, although τ-*way* CT is better than the others testing methods in hitting τ'-MFS ($\tau' \leqslant \tau$), giving from 10% to 30% differences in combination coverage, diversity and RFD when $\tau' = \tau$, but the differences become small when $\tau' < \tau$. In the S_n and S_c experiments, CT (offline CT and online CT) is better than ART (ART(SD) and ART(HD)), while ART is better than RT. However, when $\tau' > \tau$, it is hard to identify a best technique. This explains in part why some researchers observe no significant difference between RT and CT.

When applying CT, the next test case is selected from a covering array or generated by some greedy heuristics. When applying ART, the next test case is the best one from a candidate set. Although the candidate set for ART is generated randomly, when the size of candidate set is large enough and the fitness function is well designed, the effectiveness of ART can approach that of CT, and sometimes even better. As RT just selects test cases randomly, almost without any guideline, its potential of hitting MFS is relatively lower. But the abilities of RT, ART and CT in detecting high order MFS are similar.

ART's effectiveness depends on its implementation; ART with Schema Distance is much better than the traditional ART with Hamming Distance in hitting MFS. From the perspective of combination coverage, ART(SD) is employing the same objective as CT, but using a different method to select candidate test cases. ART effectively degrades into RT when the newly generated test case is just slightly different from all executed test cases.

RT, ART and CT play different roles, and complement each other. Based on our review and experiments, we strongly recommend CT in hitting MFS, as it is particularly designed for this. There remain issues for investigation:

1. Existing test suite generation methods for CT, especially greedy methods and search based methods, use τ-*way* combination coverage for fitness. Using RFD for fitness, we may produce a better prioritized test suite; in our experience, RFD is the best metric for measuring the MFS detecting ability.
2. More evidence should be collected on the effectiveness of RT, ART and CT, not just from the simulation experiments with mutated faults, but also from real-life software with real faults.
3. ART can use search-based techniques for its fitness function, such as Simulated Annealing, hill-climbing, and the like. They could also use different greedy techniques like IPO (In Parameter Order), DDA (Deterministic Density Algorithm), and others. Using alternative methods could improve their ability to hit the MFS.
4. After hitting the MFS, it remains a problem to identify the MFS, as a test case of SUT includes many different value schemas. Finding out which of them triggers the failure is an important problem that can facilitate debugging effort by reducing the scope of the code that needs to be inspected.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 61272079), the Research Fund for the Doctoral Program of Higher Education of China (No. 20130091110032), the Science Fund for Creative Research Groups of the National Natural Science Foundation of China (No. 61321491), and the Major Program of National Natural Science Foundation of China (No. 91318301).

References

[1] C. Nie, H. Leung, The minimal failure-causing schema for combinatorial testing, ACM Trans. Software Eng. Meth. (TOSEM) 20 (4) (2011) 1–38.
[2] T.Y. Chen, F.-C. Kuo, R.G. Merkel, T.H. Tse, Adaptive random testing: the art of test case diversity, J. Syst. Softw. 83 (1) (2010) 60–66.
[3] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2) (2011) 1–29.
[4] A.M. Salem, A Software Testing Model: Using Design of Experiments (DOE) and Logistic Regression, Ph.D. Thesis, Florida Institute of Technology, Melbourne, Florida, December 2001.

[5] D.R. Kuhn, D.R. Wallace, Software fault interactions and implications for software testing, IEEE Trans. Softw. Eng. 30 (6) (2004) 418–421.
[6] P.J. Schroeder, P. Bolaki, V. Gopu, Comparing the Fault Detection Effectiveness of n-Way and Random Test Suites, in: ISESE, IEEE Computer Society, 2004, pp. 49–59.
[7] M. Ellims, D. Ince, M. Petre, The effectiveness of t-way test data generation, in: Proceedings of the 27th International Conference on Computer Safety, Reliability, and Security, SAFECOMP '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 16–29.
[8] A. Arcuri, L. Briand, Formal analysis of the probability of interaction fault detection using random testing, IEEE Trans. Software Eng. 38 (5) (2011) 1088–1099, http://dx.doi.org/10.1109/TSE.2011.85.
[9] T.Y. Chen, H. Leung, I. Mak, Adaptive random testing, in: Proceedings of the Ninth Asian Computing Science Conference (ASIAN04), Springer, Berlin/Heidelberg, 2004, pp. 320–329.
[10] J. Bach, P.J. Schroeder, Pairwise testing – a best practice that isn't, in: Proc. 22nd Pacific Northwest Software Quality Confer, 2004, pp. 180–196.
[11] S.R. Dalal, C.L. Mallows, Factor-covering designs for testing software, Technometrics 40 (3) (1998) 234–243.
[12] S. Elbaum, G. Rothermel, S. Kanduri, A.G. Malishevsky, Selecting a cost-effective test case prioritization technique, Software Quality Control 12 (2004) 185–210.
[13] G. Rothermel, R. Untch, C. Chu, M. Harrold, Prioritizing test cases for regression testing, IEEE Trans. Software Eng. 27 (10) (2001) 929–948, http://dx.doi.org/10.1109/32.962562.
[14] M. Cohen, P. Gibbons, W. Mugridge, C. Colbourn, J. Collofello, Variable strength interaction testing of components, in: Computer Software and Applications Conference, 2003, COMPSAC 2003, Proceedings, 27th Annual International, IEEE Computer Society, Dallas TX, 2003, pp. 413–418. http://dx.doi.org/10.1145/1273463.1273482.
[15] D.R. Kuhn, M.J. Reilly, An investigation of the applicability of design of experiments to software testing, in: SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02), IEEE Computer Society, Washington, DC, USA, 2002, p. 91.
[16] D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, The combinatorial design approach to automatic test generation, IEEE Softw. 13 (5) (1996) 83–88. http://dx.doi.org/10.1109/52.536462.
[17] Z.Q. Zhou, Using coverage information to guide test case selection in adaptive random testing, in: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, COMPSACW '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 208–213.
[18] T.Y. Chen, D.H. Huang, F.-C. Kuo, R.G. Merkel, J. Mayer, Enhanced lattice-based adaptive random testing, in: Proceedings of the 20 09 ACM symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 422–429.
[19] A. Arcuri, L. Briand, Adaptive random testing: an illusion of effectiveness?, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, ACM, New York, NY, USA, 2011, pp 265–275.
[20] G.J. Myers, C. Sandler, T. Badgett, T.M. Thomas, The art of software testing second edition, in: The Art of Software Testing Second Edition, John Wiley and Sons, New Jersey, 2004, pp. 3–10.
[21] J.W. Duran, S.C. Ntafos, An evaluation of random testing, IEEE Trans. Software Eng. (1984) 438–444.
[22] T.A. Thayer, Software reliability, Carnegie-Rochester Conference Series on Public Policy, North-Holland, Amsterdam, 1978.
[23] T.Y. Chen, F.-C. Kuo, H. Liu, Distributing test cases more evenly in adaptive random testing, J. Syst. Softw. 81 (2008) 2146–2162.
[24] R.M. Parizi, A.A.A. Ghani, R. Abdullah, R. Atan, On the applicability of random testing for aspect-oriented programs, Int. J. Softw. Eng. Appl. 3 (4) (2009) 1–20.
[25] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 75–84.
[26] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Trans. Softw. Eng. Meth. 5 (1996) 63–86.
[27] W. Visser, C.S. Pasareanu, R. Pelánek, Test input generation for java containers using state matching, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA '06, ACM, New York, NY, USA, 2006, pp. 37–48.
[28] D. Hamlet, R. Taylor, Partition testing does not inspire confidence, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988, 1988, pp. 206–215, http://dx.doi.org/10.1109/WST.1988.5376.
[29] E.J. Weyuker, B. Jeng, Analyzing partition testing strategies, IEEE Trans. Softw. Eng. 17 (1991) 703–711.
[30] W. Gutjahr, Partition testing vs. random testing: the influence of uncertainty, IEEE Trans. Software Eng. 25 (5) (1999) 661–674, http://dx.doi.org/10.1109/32.815325.
[31] A. Arcuri, M.Z. Iqbal, L. Briand, Formal analysis of the effectiveness and predictability of random testing, in: ISSTA '10: Proceedings of the 19th International Symposium on Software Testing and Analysis, ACM, New York, USA, 2010, pp. 219–230. http://dx.doi.org/10.1145/1831708.1831736.
[32] F.T. Chan, T.Y. Chen, I.K. Mak, Y.T. Yu, Proportional sampling strategy: guidelines for software testing practitioners, Inf. Softw. Technol. 38 (12) (1996) 775–782.
[33] K.P. Chan, T.Y. Chen, D. Towey, Good random testing, in: Ada-Europe, 2004, pp. 200–212.
[34] B. Zhou, H. Okamura, T. Dohi, Markov chain monte carlo random testing, in: Proceedings of the 20 10 International Conference on Advances in Computer Science and Information Technology, AST/UCMA/ISA/ACN'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 447–456.
[35] H. Hemmati, A. Arcuri, L. Briand, Reducing the cost of model-based testing through test case diversity, in: Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, ICTSS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 63–78.
[36] F.-C. Kuo, K.Y. Sim, C. ai Sun, S.-F. Tang, Z. Zhou, Enhanced random testing for programs with high dimensional input domains, in: SEKE, 2007, pp. 135–140.
[37] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: Proceeding of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 1–10.
[38] T.Y. Chen, R. Merkel, An upper bound on software testing effectiveness, ACM Trans. Softw. Eng. Meth. 17 (2008) 16:1–16:27.
[39] R. Mandl, Orthogonal latin squares: an application of experiment design to compiler testing, Commun. ACM 28 (10) (1985) 1054–1058.
[40] D. Kuhn, R. Kacker, Y. Lei, Random vs. combinatorial methods for discrete event simulation of a grid computer network, in: Proceedings, Mod Sim World 2009, October 14–17, 2009, pp. 83–88.
[41] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino, Applying design of experiments to software testing: experience report, in: ICSE '97: Proceedings of the 19th International Conference on Software Engineering, ACM, New York, NY, USA, 1997, pp. 205–215. http://dx.doi.org/10.1145/253228.253271.
[42] W.K. Ehrlich, V.N. Nair, D.A. James, J. Zevallos, A statistical assessment of some software testing strategies and application of experimental design techniques, Stat. Sinica 8 (1998) 165C184.
[43] N. Kobayashi, T. Tsuchiya, T. Kikuno, Non-specification-based approaches to logic testing for software, Inf. Softw. Technol. 44 (2) (2002) 113–121.