This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing

Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia and Mark Harman

Abstract—We present an empirical comparison of three test generation techniques, namely, Combinatorial Testing (CT), Random Testing (RT) and Adaptive Random Testing (ART), under different test scenarios. This is the first study in the literature to account for the (more realistic) testing setting in which the tester may not have complete information about the parameters and constraints that pertain to the system, and to account for the challenge posed by faults (in terms of failure rate). Our study was conducted on nine real-world programs under a total of 1683 test scenarios (combinations of available parameter and constraint information and failure rate). The results show significant differences in the techniques' fault detection ability when faults are hard to detect (failure rates are relatively low). CT performs best overall; no worse than any other in 98% of scenarios studied. ART enhances RT, and is comparable to CT in 96% of scenarios, but its computational cost can be up to 3.5 times higher than CT when the program is highly constrained. Additionally, when constraint information is unavailable for a highly-constrained program, a large random test suite is as effective as CT or ART, yet its computational cost of test generation is significantly lower than that of other techniques.

Index Terms—combinatorial testing; random testing; adaptive random testing;

1 Introduction

Combinatorial Testing (CT) is a potentially powerful technique for revealing faults in software systems. However, one of the key inputs to the approach is the model of the parameters and constraints that pertain to the system. Previous studies have assumed perfect knowledge of all parameters and constraints. This may be sadly unrealistic in many practical real-world scenarios, because modelling is still a labour intensive and error-prone process with neither automatic techniques nor general rules on which testers can rely [1]. Thus a practising tester may not be able to determine all the parameters that may affect their system, nor all the constraints that delimit valid potential parameter combinations [2], [3].

In the absence of constraints, it is known that purely random testing is surprisingly effective at achieving combinatorial coverage [4], [5]. This raises the natural question as to the relative performance of combinatorial testing techniques against random (and improved adaptive versions of pure random testing), when the tester has imperfect (partial) knowledge of the constraints and parameters that apply.

Furthermore, a fault that is easy to detect (because many test cases execute the fault and cause it to lead to failure) may have different characteristics from those faults that are harder to detect. Practising testers may be interested in different kinds of faults (as characterised by their likely failure rate) for different systems. For example, a well-tested system that has proved reliable may contain a high proportion of hard-to-detect faults, characterised by lower failure rates, while a new implementation of a poorly understood system may have a higher proportion of easier-to-detect (higher failure rate) faults. Therefore, it is important for practicing testers to understand the differences in the behaviour of testing techniques as the failure rate of the faults present in the system varies.

We study three testing techniques in this work: combinatorial testing (CT), also known as combinatorial interaction testing (CIT), random testing (RT) and adaptive random testing (ART) in the presence of partial information about parameters and constraints and with respect to different levels of fault 'challenge' (denoted by faults' failure rates). CT, RT and ART are all popular testing techniques in practice [1], [6]. They require no knowledge about the implementation of software and their test suite generations can all be automated [7]. In order to choose between these three techniques, one needs to understand their relative effectiveness and efficiency. To this end, many previous comparative studies have been reported.

Table 1 summarises previous studies of CT, RT and ART. When comparing CT against RT, previous studies often lead to controversial results: some suggested that CT is more effective than random test suite of similar or larger size [8], [9], [10], [11], [12], [17], [18], [21], while others found that the difference between CT and RT is not as large as people expect [13], [14], [15], [16], [19], [20]. Most of these previous studies are derived from a few example cases; only two ([20] and [21]) reported empirical results beyond case analysis.

Regarding comparisons between ART and RT, it is generally agreed that ART enhances RT, which was shown

- *H. Wu and C. Nie are with Department of Computer Science and Technology, Nanjing University, Nanjing, China, 210023. E-mail: hywu@smail.nju.edu.cn, changhainie@nju.edu.cn*
- *J. Petke is with CREST, Computer Science, University College London, London, UK, WC1E 6BT. E-mail: j.petke@ucl.ac.uk*
- *Y. Jia and M. Harman are with Facebook Inc., London, UK, W1T 1FB and CREST, Computer Science, University College London, London, UK, WC1E 6BT. E-mail: {yue.jia, mark.harman}@ucl.ac.uk*

Manuscript received XX XX, XXXX; revised XX XX, XXXX.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

TABLE 1
Overview of literature on the comparisons of CT, RT and ART.

Reference	Type of Experiment	Constraints Included?	CT / RT	ART / RT	CT / ART
[8], [9], [10], [11], [12], [13], [14], [15], [16]	Case Analysis	No	√		
[17], [18], [19]	Simulation	No	√		
[20], [21]	Empirical Study	Yes	√		
[5]	Formal Analysis	No	√		
[22]	Simulation	No		√	
[23], [24], [25], [26]	Empirical Study	No		√	
[27]	Simulation	Yes			√
[28]	Simulation	No	√	√	√

both theoretically and empirically [22], [23], [24], [25], [26]. However, these studies assumed no constraints between parameters, which may be problematic as real-world applications often have constrained domains [29], [30]. There is only one previous study [28] that compared all three (CT, RT and ART), but only for synthetically-generated test models (all models are unconstrained and every synthetic fault is caused by an exact k-way combination), thereby reducing the real-world relevance of its findings. Moreover, none of the above studies has taken different test scenarios into account, whereas the behaviour of CT, RT and ART might change as the test scenario changes.

In order to better understand the benefits and limitations of CT, RT and ART under different test scenarios, a further empirical comparison is thus needed. The study of partial parameters and constraints is important, because it increases relevance to real-world test scenarios. It is clearly optimistic to expect that, in all cases, a tester will be aware of all such parameters and constraints, potentially artificially inflating the perceived performance of different testing techniques by imbuing them with unrealistic levels of information. Therefore, we adopt a more nuanced approach in which we evaluate testing techniques in the presence of imperfect information, allowing us to report on the performance of the testing techniques when the tester has incomplete information about either parameters or constraints (or both).

Intuition might suggest that CT would be a good choice when the fault is hard to detect in a complex software system [31], but previous studies do not provide quantitive results in terms of either fault proneness or search space size. Additionally, ART might be suggested as an alternative technique to CT as a large proportion of combinations among parameters is expected to be covered by a similarity-based heuristic for test suite generation [27], but this finding has not yet been empirically evaluated. Moreover, with the many different test scenarios in real-world applications, it is important to investigate how the performance of CT, RT and ART changes as the test scenario changes.

The study we report here aims to investigate these questions more thoroughly. Specifically, this is the first to account for different test scenarios on the relative performance of CT, RT and ART. We conducted our experiment on nine real-world programs: six of them are from the Software-artifact Infrastructure Repository (SIR) [32]. The other three are widely-used relatively larger highly-configurable programs [21], [33]. By combining different choices of the three features that form our test scenario (i.e. proportion of parameters available, proportion of constraints available and failure rate), a total of 1683 test scenarios was created. For each test scenario, we applied CT, RT and ART to generate test suites of the same size, and evaluated their relative performance in terms of fault detection ability and computational cost. This study seeks to provide not only quantitative data to underpin our intuitions about CT, RT and ART, but also a baseline against which others can extend and explore these relationships in future work.

The findings of this study provide suggestions on the choices of CT, RT and ART with respect to different test scenarios. Specifically, our primary findings are as follows:

1) Clearly *any* techniques will become indistinguishable below very low and also above very high failure rates; low rates denote almost-impossible-to-find faults, whereas faults with very high failure rates would allow any non-trivial technique to expose them. Our study gives quantitative bounds to these intuitions, and does so for CT, RT and ART. Specifically, we found that the techniques become indistinguishable at failure rates below 0.0001 and above 0.71. We also observed that the approximate 'sweet spot of differentiation' (at which techniques are maximally distinguishable) lies between failure rates of roughly 0.001 and 0.2.
2) Overall, CT is recommended as the most favourable technique; its fault detection ability is no worse than RT and ART in 98% of test scenarios, especially when the failure rate is lower than 0.005, and all constraints are present in the model. However, when no constraints are available in the model, the three testing techniques tend to perform equally well, which makes RT more computational cost-effective.
3) As an improvement of RT, ART enhances RT in almost all test scenarios. In 96% of scenarios ART is as effective as CT, and sometimes even performs better. However, ART might be significantly slower (up to 3.5 times in our study) than CT when there is a large number of constraints present in the model and all parameters are involved in constraints.
4) A high strength (large test suite) is desirable when only partial constraint information is available, where the average difference in proportion of faults detected between 4-way and 2-way testing is 0.25. A high strength is also much more desirable when the failure rate is

TABLE 2
A test model for 'font effect'.

Font style	Font size	Underline	Superscript	Subscript
Regular	5	On	On	On
Italic	12	Off	Off	Off
Bold	20			

hard constraint: *Superscript* = On $\wedge$ *Subscript* = On

between 0.001 and 0.05, where up to 89% of faults can be further detected by a sufficiently large test suite.

5) When constraint information is unavailable for a highly constrained program, a large random test suite is preferable, because it performs as well as CT and ART but has a very low computational cost. By contrast, when all constraints are present in the model, we recommend using a small but systematically-designed test suite, such as a 2-way CT, at first, and then increase strength as needed.

The rest of this paper is organised as follows. Section 2 introduces basic concepts and the three test suite generation techniques: CT, RT and ART. Section 3 describes our research questions and experimental design. Section 4 reports the results. Section 5 discusses some implications of our findings. Section 6 describes threats to validity. Section 7 presents related work, and Section 8 concludes this paper.

2 Background

We first introduce the notation used throughout the paper and describe the three test case generation techniques under study: combinatorial testing (CT), random testing (RT) and adaptive random testing (ART).

2.1 Test Model and Test Suite Generation

Software behaviour is often impacted by its parameters and their interactions. Suppose the system under test (SUT) has n parameters. These could represent configuration parameters, internal or external events, user inputs etc. Let each parameter have l_i discrete values from the finite set V_i. A test case is a tuple $t = (x_1, x_2, \ldots, x_n)$ with $x_i \in V_i$ for $1 \leq i \leq n$, and $T_{all} = V_1 \times V_2 \times \ldots \times V_n$ denotes the search space which consists of all possible test cases. In modern software systems, the size of T_{all} is usually prohibitively large [31], and so the tester needs to use some strategies to automatically sample a subset of test cases for testing.

Additionally, not all combinations of parameters can be valid due to constraints between parameters in the SUT. An example of a constraint is $(a_1 \wedge b_2)$, stating that parameter assignments a_1 and b_2 cannot be combined together. Constraints will prevent any test cases containing them from execution, and so constraint handling strategies should be applied during test suite generation. Constraints can be classified as hard or soft constraints [34]. Hard constraints state that certain parameter combinations cannot appear in any test case. Soft constraints, on the other hand, only indicate that these combinations need not be covered. These are usually identified by software testers and cover interactions that, for example, rarely occur in practice.

TABLE 3
A 2-way covering array $CA(9; 2, 3^2 2^3)$.

	Font style	Font size	Underline	Superscript	Subscript
t_1	Regular	5	On	On	Off
t_2	Regular	12	Off	Off	Off
t_3	Regular	20	Off	Off	On
t_4	Italic	5	On	Off	On
t_5	Italic	12	Off	On	Off
t_6	Italic	20	Off	On	Off
t_7	Bold	5	Off	On	Off
t_8	Bold	12	On	Off	On
t_9	Bold	20	On	Off	Off

Parameters, their corresponding values and constraints form a test model. Table 2, for example, shows a test model for testing font effects in a word processor. This model has $n = 5$ parameters with $|V_1| = |V_2| = 3$ and $|V_3| = |V_4| = |V_5| = 2$. There exists one hard constraint: a piece of text cannot be set as a superscript and a subscript at the same time.

Algorithm 1 One-test-at-a-time Framework

1: $T = \emptyset$
2: **while** stop condition is not met **do**
3: Generate a constraint satisfying test case t according to the given criterion
4: $T = T \cup t$
5: **end while**
6: **return** T

In order to generate a test suite for a given test model, a common strategy is the one-test-at-a-time framework. Algorithm 1 illustrates its top level process. This framework starts with an empty test suite, and then test cases are generated iteratively until the given stopping condition is met. We used this framework because it can be adapted to any of the three test suite generation techniques considered (CT, RT and ART), and the only difference between them lies in the selection strategy of the next test case (Line 3). As there are no available RT and ART tools that support constraint handling, we implemented all generation algorithms to avoid potential environmental bias.

2.2 Combinatorial Testing

The key insight of combinatorial testing (CT), or combinatorial interaction testing (CIT), is that not every parameter assignment triggers a fault, but it is the interactions between various parameters that lead to software failures [1]. A CT test suite, i.e. τ-way covering array, is designed to only cover the combinations among a fixed number of τ parameters in order to test parameter interactions up to 'strength' τ. Empirical studies [31] have demonstrated that the number of parameters that lead to a software failure is usually one or two, and not likely to exceed six. Therefore, CT is able to detect a large proportion of faults by applying a relatively small value of τ in practice.

In CT, the parameter interaction can be represented as a τ-way combination, i.e. a combination of τ parameter

values. The test generation strategy of CT is to construct a τ-way covering array that covers all τ-way combinations, where τ is referred to as the covering strength. A covering array is represented as $CA(N; \tau, {l_1}^{g_1} {l_2}^{g_2} \ldots {l_k}^{g_k})$, where N is the size of the array, and ${l_i}^{g_i}$ represents g_i parameters with the same number of l_i values.

For example, a 2-way covering array for the test model in Table 2 can be represented as $CA(9; 2, 3^2 2^3)$, as shown in Table 3. Instead of examining all possible $3^2 \times 2^3 = 72$ test cases, CT only needs 9 test cases to cover every valid 2-way combination at least once. If interactions between no more than two parameters trigger a fault, then the effectiveness of a 2-way covering array is equivalent to exhaustive testing.

To generate a τ-way covering array, the criterion used in Algorithm 1 is to generate a test case that covers the largest number of yet uncovered τ-way combinations. We implemented a variant of AETG [35] as it is an effective algorithm for one-test-at-a-time constrained covering array generation. At each iteration, the algorithm firstly reorders parameters randomly and then assigns values to parameters one after another. For each parameter, the number of uncovered τ-way combinations that can be covered by assigning each value is calculated, and the value that leads to the maximum coverage is assigned to this parameter. A Boolean satisfiability (SAT) solver is applied to determine whether the value assignment satisfies the constraints or not. If any invalid combinations are introduced, the parameter will be reassigned to another value that does not violate constraints. The above process is repeated until all valid τ-way combinations are covered, and finally a τ-way covering array is returned.

Aside from AETG, researchers have also proposed many other algorithms for covering array generation. These include algebraic approaches [36], Constraint Satisfaction Problem (CSP) solving [37], incremental Boolean satisfiability (SAT) solving [38], hyper-heuristic search [39] and two-mode search [40]. There are also tools, such as PICT [41], ACTS [42] and CASA [43]. A comprehensive survey of all the techniques can be found in [44].

2.3 Random Testing

Instead of intentionally sampling test cases to target particular kinds of faults, random testing (RT) generates test inputs at random. This technique is conceptually simple and easy to implement, yet it has shown to be useful [45], and is one of the few testing techniques whose fault detection ability has been theoretically analysed [4], [5]. In our study we use RT to generate test cases until a test suite of given size is generated. The test suite size is determined according to the corresponding CT test suite, i.e. the size of a τ-way covering array.

In order to generate a RT test suite, the criterion used in Algorithm 1 is to generate a random test case by assigning random values to each parameter. Similarly, to handle constraints, a SAT solver is used to determine whether each value assignment satisfies the constraints. If it introduces any invalid combinations, the parameter will be reassigned to another random value.

2.4 Adaptive Random Testing

Based on the observation that fault causing inputs are often clustered into contiguous regions, adaptive random testing (ART) was proposed to enhance RT by spreading randomly sampled test cases over the whole search space as evenly as possible [23]. The goal of ART is to promote diversity, or in other words, dissimilarity of test cases: if a test case does not trigger a fault, a better sampling strategy for the next test case is to make it as 'far away' as possible from the previously executed test cases [6]. In our study we also use ART to generate test suites of the same size as the size of corresponding CT suites. ART can be thought of as a form of Search Based Software Testing (SBST) [46], in which the fitness function is diversity.

In order to generate an ART test suite, the criterion used in Algorithm 1 is to generate a test case that is as different as possible from the previously generated ones. We implemented the Fixed-Size-Candidate-Set ART algorithm (FSCS-ART) [23] as it is one of the best ART algorithms [7]. At each iteration, the algorithm firstly generates M candidate test cases by random testing (in this work we let $M = 30$), where a SAT solver is used to ensure that each of these M candidates satisfies the constraints. Subsequently, a distance (or similarity) based metric is used to evaluate these candidates. The best test case is the one that has the largest distance from the previously generated test suite, where the distance between a test case t and a test suite T is defined as the minimum Hamming distance between t and each test case in T.

3 Empirical Study Design

We describe the experiments we conducted in order to answer the research questions posed below.

3.1 Research Questions

Our goal is to compare three testing techniques for highly-configurable systems: combinatorial testing (CT), random testing (RT) and adaptive random testing (ART). We empirically show their fault detection ability and computational cost under different test scenarios in order to better understand their benefits and limitations.We aim to answer the following research questions:

- $\mathbf{RQ_1}$ Under which test scenarios is there a significant difference in performance between the three testing techniques?
- $\mathbf{RQ_2}$ How effective are the three testing techniques investigated at detecting faults under different test scenarios?
- $\mathbf{RQ_3}$ How does covering strength impact the effectiveness of the three testing techniques under different test scenarios?
- $\mathbf{RQ_4}$ How efficient are the three testing techniques under different test scenarios?

3.2 Subject Programs

In this study, we used nine subject programs: FLEX, GREP, GZIP, SED, MAKE, NANOXML, DRUPAL, BUSYBOX and LINUX. Table 4 gives details of these subject programs[1].

1. All test models and corpus of faults are available in this paper's companion website: http://gist.nju.edu.cn/doc/ec18/.

TABLE 4
Details of the subject programs and the number of test scenarios generated.

Name	Description	LOC [21], [32], [33]	Model	# Parameters (n)	# Constraints	# Faults	# Scenarios
FLEX	lexical analyser	15,297	$2^2 3^2 2^4 5^1$	9	12	50	270
GREP	text-search utility	15,633	$3^2 4^1 6^1 8^1 4^1 3^1 2^1 5^1$	9	83	12	125
GZIP	compression utility	6,582	$2^{13} 3^1$	14	61	5	70
SED	stream text editor	11,148	$2^4 6^1 10^1 2^1 4^1 2^2 3^1$	11	50	22	250
MAKE	build utility	27,879	2^{10}	10	1	2	14
NANOXML	XML parser	7,646	$2^5 4^1 2^1$	7	6	16	54
DRUPAL	web framework	336,025	2^{47}	47	45	160	530
BUSYBOX	UNIX utilities	189,722	2^{68}	68	16	9	120
LINUX	operation system	12,594,584	2^{104}	104	83	28	250

The first six programs come from the Software-artifact Infrastructure Repository (SIR) [32]. These programs are all real-world, open source command line utilities. This is a now standard set of benchmarks in combinatorial testing literature [29], [47], [48], [49], [50]. The last three programs are real-world, open source, highly-configurable systems, which are much larger than SIR programs in both size and number of parameters: DRUPAL is a modular framework for web content management [33]; BUSYBOX is software that provides UNIX utilities in a single executable file [21]; and LINUX is a well-known operation system [21].

The test models of the six SIR programs have been used in previous work [29]. Each parameter of these models represents an input provided to the program, which can be a flag option (such as '-c'), a digit, a path to a file, etc. The parameters, values and constraints of each model were manually extracted from the test plan, which is described in the Test Specification Language (TSL), provided by SIR. SIR also provides files and scripts to execute its test cases.

The test model of DRUPAL comes from its feature model[2], reported in previous work [33]. Each parameter represents a module, which can be enabled or disabled to customise the functionality of the system. The parameters and constraints of the feature model were manually extracted from the software's documentation.

The test models of BUSYBOX and LINUX are extracted from their original models provided in previous work [21]. Each parameter of these models represents a configuration option implemented through conditional compilation in the C preprocessor (i.e., using *#ifdef* directive). The parameters and constraints are extracted from the KConfig file [21], [51]. The original models of BUSYBOX and LINUX are large, having 651 and 31,713 parameters, and 615 and 293,826 constraints, respectively. We found that such models cannot be efficiently handled by our greedy-based generation algorithm[3]. We are also not aware of any CT tools that can generate τ-way test suites ($\tau > 2$) for these models[4]. As our aim is to compare τ-way CT with RT and ART ($\tau = 2, 3, 4$) in a controlled experiment, we used only the files that the bug reports mentioned in order to extract sub-models for these two programs. Algorithm 2 gives the process of model extraction: we firstly determined whether each fault can be detected in the original model (a fault is undetectable if it is triggered by parameters that are not in the original model)[5]; then for those source files containing detectable faults, we extracted all relevant parameters and constraints from the original model. The numbers of selected parameters for BUSYBOX and LINUX are 68 and 104, respectively.

Algorithm 2 Process to extract sub-model

```
S = {}, P = {}, C = {}
for each fault f in the corpus of faults do
    if f can be detected in the original model then
        add all parameters in the file containing f into S
    end if
end for
for each parameter p in the original model do
    if p is a parameter in S then
        add p into P
    end if
end for
for each constraint c in the original model do
    if all parameters involved in c are also in P then
        add c into C
    end if
end for
M = create a test model based on P and C
return M
```

In this study, all constraints are 'hard' constraints. Each constraint in the test model is encoded by a Boolean formula, representing a condition that must be satisfied by any test case. The sixth column of Table 4 gives the number of constraints (i.e., the number of Boolean formulae) in

2. The parameters in our test model are independent, but the parameters in a feature model are tree-structured. Therefore, we added additional constraints to maintain the hierarchical dependency of the feature model. The input space of our test model is the same as that of the original feature model.

3. For CT, it was only possible to generate 2-way test suites for the original model of BUSYBOX. We ran out of memory (on a machine with 16GB RAM) for all other cases ($\tau = 3, 4$ for the original model of BUSYBOX, and any value of τ for the original model of LINUX).

4. A previous work [21] estimates that the generation of 3-way CT test suite for LINUX could take months and require more than 1TB RAM to track the combinations to be covered.

5. Note that the fault mining and the modelling of these programs are two separate and independent processes, so some faults are not detectable in their original models. As the result, the numbers of faults used in this study (as shown in Table 4) are smaller than those reported in the previous study [21].

each program. Different constraints may involve different numbers of parameters. As there is no evidence that a constraint with more parameters is harder for the tester to model, all constraints are treated equally.

Note that the identification of parameters, values and constraints for a given program is still an open problem in CT [1]. In this study, we used the same models (for the first seven programs), or extracted subsets of models (for BUSYBOX and LINUX), from previous works to avoid potential bias from creating models for these programs.

In order to investigate fault detection ability for each of the three testing techniques, for the six SIR programs, we considered all available software versions in SIR [32]; the test model is the same for different versions of the same program. For each version, SIR provides a series of faults, which are seeded by developers to represent bugs that they have encountered. SIR also provides the corresponding fault matrix indicating which test cases can trigger each fault. As some of these faults are triggered by parameters that are not in our test models (namely they cannot be detected by any of CT, RT and ART test suites generated in this study), for each SIR program, we exhaustively examined all possible test cases of our test models and selected only the set of faults that can be detected by at least one test case. We note that the different versions are used only to collect the number of faults, thus the faults from different versions are considered equally (we did not distinguish the version to which a fault belongs in our test scenarios).

For DRUPAL, BUSYBOX and LINUX, we used an existing corpus of real faults from previous work [21], [33], where the combinations of parameter values that can trigger each fault are provided. According to previous work [21], [33], these faults are mostly mined from bug tracking systems; they are all reported to the original developers and either confirmed and/or fixed by the developers. Thus they are all real-world faults. The main intention of selecting these three programs is to evaluate the three testing techniques on larger programs (test models) and real faults, which cannot be achieved with SIR programs alone.

The seventh column of Table 4 gives the total number of faults that are used in this study.

3.3 Test Scenarios

We consider three features of the system under test that might impact the performance of the three testing techniques considered (CT, RT and ART): proportion of parameters available ($para$) and proportion of constraints available ($cons$) in the model, and the failure rate ($rate$) of the faults to be detected (based on existing failure data). Combinations of choices of these three features $< para,\ cons,\ rate >$ form a test scenario.

The proportion of parameters available ($para$) defines the size of the raw search space. Determining a proper set of parameters and their values is a key task in modelling software systems [1]. In practice, a tester might miss some parameters during the modelling phase because of poor understanding of software specification. As a result, the parameters available to an automated testing technique might be fewer than those that are truly relevant. To investigate the impact of different sets of available parameters on the effectiveness of different testing techniques, for each program's complete model we created a series of test models with different $para$ by selecting its first $\lceil para \times n \rceil$ parameters, where n is the total number of parameters in the complete model as shown in Table 4. This way, for a given program, the larger the value of $para$, the more parameters are included in the test model, and thus the larger the raw search space to be explored. We let $para = \{0.4,\ 0.6,\ 0.8,\ 1.0\}$ for all programs. To determine whether a test case consisting of $para < 1.0$ parameters can detect a fault or not, we complemented this test case by adding other parameters with default values, so that the fault matrix and corpus of faults of the complete model can still be used.

Furthermore, the tester might also be unaware of some of the pertinent constraints that govern the space of valid inputs. The proportion of constraints available ($cons$) denotes the degree to which the constraints have been fully identified by the tester. As test cases containing any invalid combinations cannot be executed, the test suite generated according to a model with low $cons$ will lead to a large number of invalid test cases, and thus make it harder to detect faults for a given test suite size. Finding all constraints is a difficult task in practice. Considering different values of $cons$ is thus an important issue in assessing the real-world effectiveness of testing techniques that might be affected by partial constraint identification. Given a test model with $para$ parameters, we created models with different $cons$ by firstly finding all constraints that correspond to these $para$ parameters, and then selecting the first $cons$ constraints to create a new model. We let $cons = \{0,\ 25\%,\ 50\%,\ 75\%,\ 100\%\}$ for all programs except MAKE and NANOXML. For MAKE and NANOXML, as there are only one and six constraints in their complete models, we let their $cons$ be $\{0,\ 100\%\}$ and $\{0,\ 50\%,\ 100\%\}$, respectively.

The failure rate ($rate$) denotes the difficulty of detecting faults. It is often used in the measurement of effectiveness in random testing studies [23]. Given a particular test model with some $para$ and $cons$, we can determine all possible inputs and the failure causing test cases that trigger each fault. As a result, the failure rate of a fault f with respect to a test model M is defined as follows [52]:

$$rate_M(f) = \frac{\text{number of inputs of } M \text{ that can trigger } f}{\text{number of all possible inputs of } M}$$

Note that the lower value of $rate$ the fewer test cases can detect it, thus it is likely harder to be found.

A test scenario $< para,\ cons,\ rate >$ is the combination of a test model ($para$ and $cons$) and a set of faults with the same degree of fault proneness ($rate$). Note that the failure rate of a fault is dependent on the particular input space, which is defined by the test model. For some subject programs, a fault may have a low $rate$ according to a model that contains a large number of parameters. But as some default parameter values may be relevant to the failure causing combinations, it is possible that the $rate$ of the fault becomes high when only a small number of parameters is available in the model. Therefore, to correctly represent the difficulty of detecting the fault f in a test scenario, f's $rate$ should be calculated based on the particular model M of this test scenario. As a result, for a subject program, the same fault can have a different $rate$ for each test model.

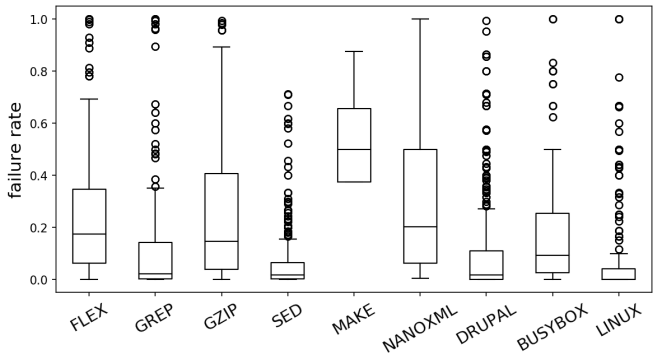

Fig. 1. The distribution of failure rates for each of the subject programs.

3.4 Process and Evaluation

In order to create test scenarios, we first created a series of test models with different combinations of $para$ and $cons$ for each subject program. Then for each test model, we determined the set of faults that can be detected by its all possible inputs, and computed the value of $rate$ for each of these faults. Note that a fault is undetectable by a model if $rate = 0$. We did not include these faults in our test scenarios, as these aren't in the search space of the model as defined by $para$ and $cons$. Finally, by combining all possible choices of $para$, $cons$ and $rate$, we got a set of test scenarios for each program. The total number of test scenarios investigated is 1683. The number of test scenarios for each subject is shown in the last column of Table 4.

For example, for GREP, where $n = 9$, we have $para = \{0.4,\ 0.6,\ 0.8,\ 1.0\}$ (which represents 4, 6, 8, 9 parameters) and $cons = \{0,\ 25\%,\ 50\%,\ 75\%,\ 100\%\}$. By combining all 4×5 choices of $para$ and $cons$, 20 test models were created. For the model with $para = 0.4$ and $cons = 0$, there are 8 out of 12 faults that can be detected. Seven of these have failure rate 0.02, while one has failure rate 0.01. Therefore, two test scenarios $< 0.4,\ 0,\ 0.01 >$ and $< 0.4,\ 0,\ 0.02 >$ were created. These are two of the many possible scenarios for GREP, which contain seven and one faults to be detected, respectively. By iterating all 20 test models and considering all failure rates for each model, we created a total of 125 test scenarios for GREP.

Figure 1 shows the distribution of all failure rates investigated for each of subject programs. For example, the second box contains all failure rates obtained from 125 scenarios of GREP. We note that most faults have a relatively low failure rate, frequently less than 0.2. Our study also includes some faults that are easy to detect, with failure rates above 0.5.

In order to evaluate fault detection ability of combinatorial testing (CT), random testing (RT) and adaptive random testing (ART), for each test scenario, we first apply CT to generate a τ-way covering array for $\tau = 2, 3, 4$, and then RT and ART to generate test suites of the same size. The faults that can be detected by each test suite are collected, and the computational cost of each testing technique is recorded. Note that the number of faults contained in each scenario can be different, so we consider the effectiveness of a testing technique for a test scenario in terms of:

$$effectiveness = \frac{\text{number of faults detected}}{\text{number of all faults in the scenario}}$$

The higher the value of this ratio the better a testing technique performs for that particular scenario. A ratio that is close to 1.0 means that the technique has a high probability to detect the fault of failure rate $rate$ by the test suite generated according to the model with $para$ and $cons$.

The three testing techniques all involve some level of randomness. Therefore we need to use inferential statistical analysis in order to cater for the inherent randomness in the algorithms [53], [54]. To collect a sample of data point from the population of all possible executions, the test suite generation and evaluation process is repeated 50 times. For each test scenario, we applied Tukey's HSD (Honest Significant Difference) test (5% significance level) [55] to distinguish the techniques that are significantly different from each other. Tukey's HSD test is a multiple comparison test. It reports a p-value for each pair of CT, RT and ART, indicating statistical significance of the difference in the proportion of faults detected. It also assigns grouping letters to each of these techniques, with "a" representing the group with the best performance, and the techniques sharing a grouping letter are not significantly different from each other. As a result, the three testing techniques are ranked into categories based on their effectiveness.

In addition, as statistical significance does not imply practical significance, we also applied Vargha and Delaneys $\widehat{A}_{12}$ statistic [56] in order to investigate the magnitude of the difference for each pair of the techniques. This measure denotes the probability that one technique outperforms another. $\widehat{A}_{12} = 0.5$ suggests that two techniques are equivalent, and the greater the $\widehat{A}_{12}$ the higher probability that the first technique yields higher values.

All experiments were carried out on a machine with Intel Xeon E5-2640 2.0 GHz CPU, 16 GB memory and CentOS 6.5 operating system.

4 Results

Next we present the results of our experiments and answer the research questions posed in Section 3.1.

We investigate the impact of different test scenarios from two viewpoints: test model and failure rate. In the real world, the tester determines the choices of $para$ and $cons$ to create the test model, yet he or she cannot know the failure rate before testing commences. Therefore, from the test model's viewpoint, we analyse the result with respect to each combination of $para$ and $cons$. In this case, the fault detection ability is evaluated based on all scenarios that have the same $para$ and $cons$. On the other hand, the failure rate alone determines the degree of the proneness of a fault. A lower/higher failure rate always indicates that the fault is harder/easier to detect, regardless of the choices of $para$ and $cons$. So from the failure rate's viewpoint, we analyse results with respect to each possible value of $rate$. We provide all experimental data for each test scenario and subject program on this paper's companion website: http://gist.nju.edu.cn/doc/ec18/.

4.1 RQ_1: Existence of Difference

The first research question asks when the three testing techniques perform differently. To answer this question, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

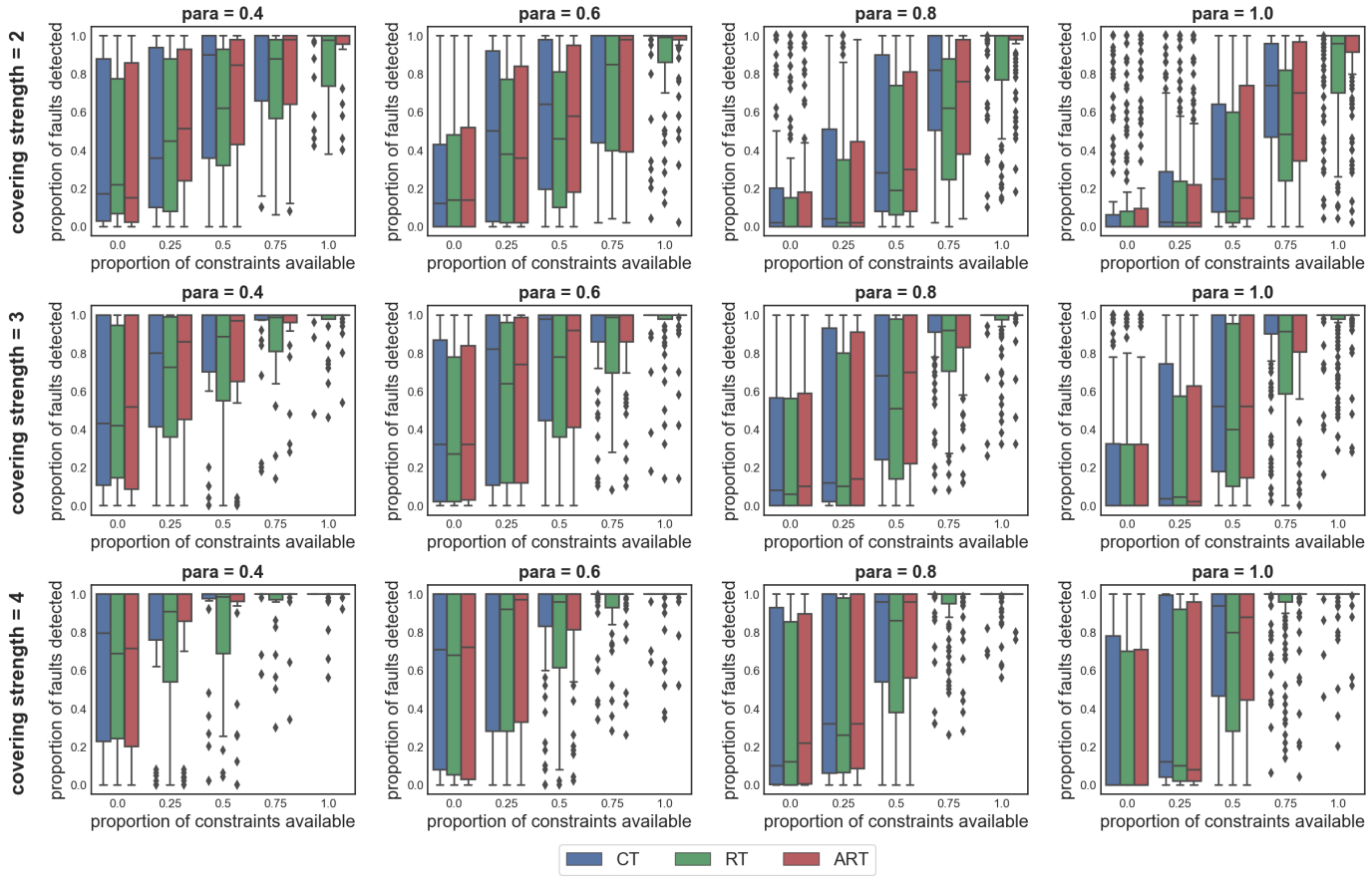

Fig. 2. Proportion of faults detected by combinatorial (CT), random (RT) and adaptive random (ART) testing under different values of $para$ and $cons$.

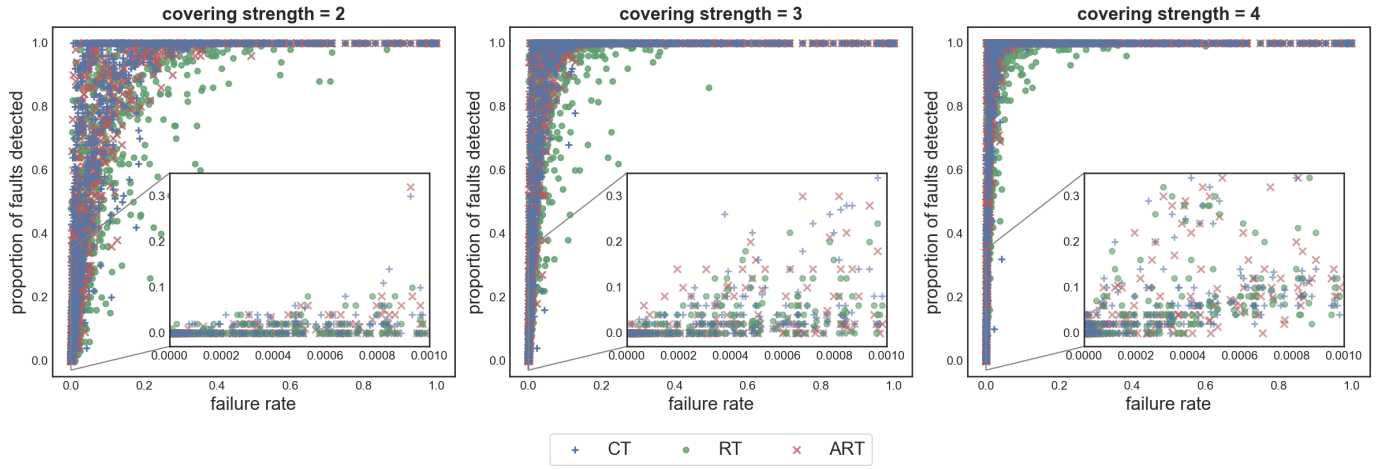

Fig. 3. Proportion of faults detected by combinatorial (CT), random (RT) and adaptive random (ART) testing under different values of $rate$.

firstly investigated the impact of different test scenarios on the effectiveness of CT, RT and ART. Secondly, we identified test scenarios that make the three testing techniques significantly different according to the statistical test, and investigated their distributions.

4.1.1 Fault Detection Ability

For each test scenario, we calculated the mean of proportions of faults detected (i.e., mean *effectiveness*) among 50 runs for each of the three testing techniques. Figure 2 shows the distribution of these means with respect to each combination of $para$ and $cons$ for covering strength (τ) 2, 3 and 4, respectively (namely each boxplot contains the means obtained from all scenarios that have a given combination of $para$ and $cons$ for a particular covering strength).

From Figure 2, we conclude that as the proportion of constraints available ($cons$) increases, the ability of detecting faults tends to increase for all three testing techniques,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

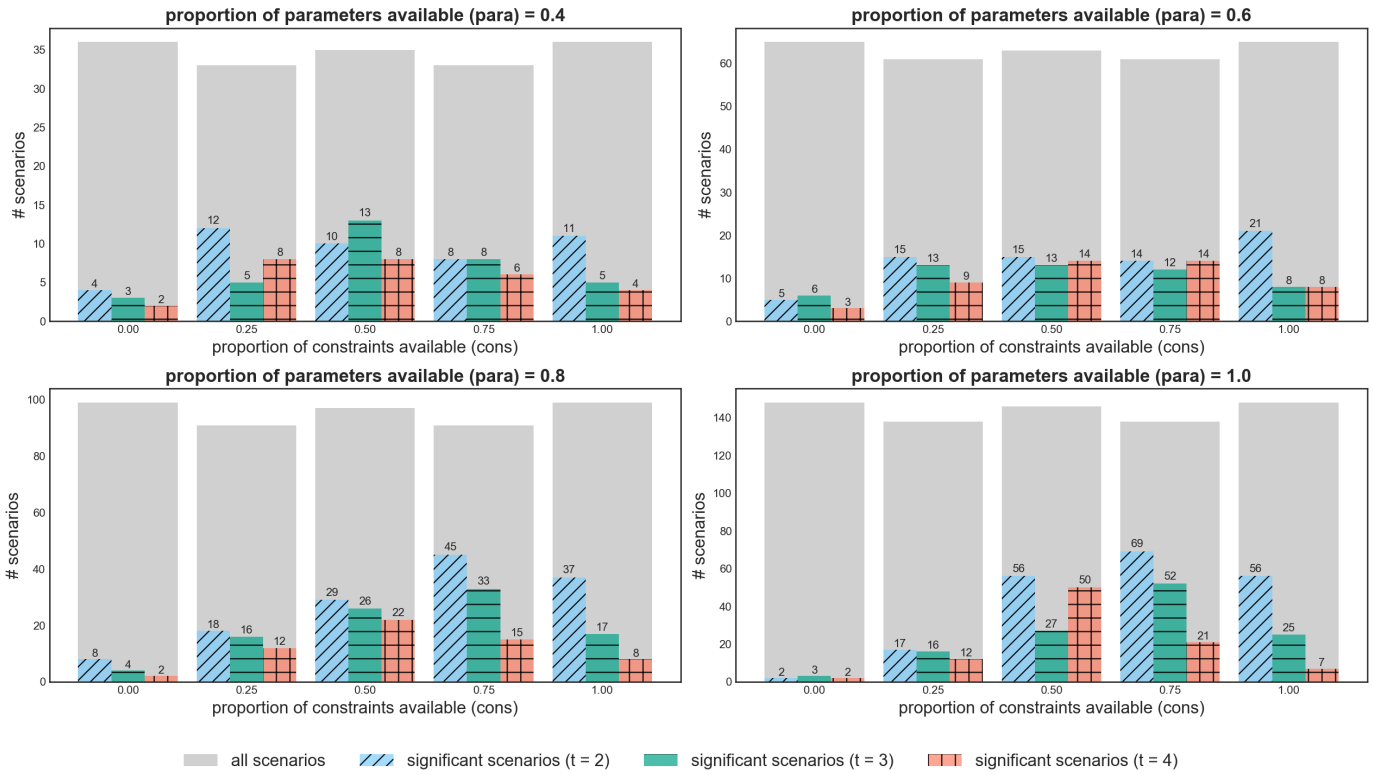

Fig. 4. Distribution of significant scenarios under different values of $para$ and $cons$.

regardless of the choices of $para$. There is also less variance in the fault detection ability, decreasing from 0.16 (for the scenarios with no constraints available in the model) to 0.04 (for the scenarios with all constraints available) for $\tau = 2$ and $para = 0.4$. This finding indicates the importance of identifying constraints for improving fault detection for all three testing techniques.

In addition, when more parameters are available in the model, the three testing techniques have a lower chance of detecting faults available in the search space. Especially for $cons = 0$, the medians of proportion of faults detected are all 0.0 for the scenarios with all parameters available ($para = 1.0$) under all covering strengths. However, when $cons = 1.0$, increasing the proportion of parameters available will not greatly reduce the chance of detecting faults. For example, when $cons = 1.0$, the faults can always be detected by CT in 75% of scenarios (the first quartile) for all choices of $para$ under all covering strengths.

Furthermore, the proportion of faults detected increases with increase in covering strength. For example, when $para = 0.4$ and $cons = 0$, the median of proportion of faults detected increase from 0.18 for covering strength 2 to 0.74 for covering strength 4. Note that higher covering strength requires more test cases. Therefore, the chance of detecting faults increases for all three testing techniques. This shows the importance of higher-strength combinatorial testing.

Figure 3 shows the average proportion of faults detected with respect to different choices of $rate$ as defined in our test scenarios. From Figure 3, we conclude that as faults with high failure rate are easy to detect, a higher $rate$ leads to better fault detection. In particular, a fault can almost always be detected, even for covering strength 2, when its failure rate is higher than 0.4. For the hardest to detect faults (with $rate$ below 0.001), the three testing techniques are not likely to detect it for covering strength 2, but their performance increases with the covering strength. Again we attribute it the fact that higher-strength test suites contain more test cases and thus have a higher chance of detecting the fault.

4.1.2 Significant Scenarios

Figures 2 and 3 illustrate the general trends concerning the fault detection ability of CT, RT and ART. Due to their randomness, we need to determine whether the difference among them is significant. A test scenario is deemed 'significant' if Tukey's HSD test determines there exists significant difference in terms of fault detection effectiveness between at least one pair of these three testing techniques in that scenario.

We used Tukey's HSD test to investigate 1683×3 scenarios (for $\tau = 2, 3, 4$) and found that 984 (19%) of these are significant. In particular, the total number of significant scenarios for each covering strength 2, 3 and 4 is 452, 305, and 227, respectively. This further confirms previous observations. As higher covering strength usually requires larger test suites, the performance differences tend to diminish with increasing test suite size.

Figures 4 shows the numbers of both significant and all scenarios under different values of $para$ and $cons$, where the results of different covering strengths are represented by different bars. For example, in Figure 4, there are 36 scenarios with $para = 0.4$ and $cons = 0$, and among these scenarios 4, 3 and 2 of them are significant scenarios when τ is 2, 3 and 4, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

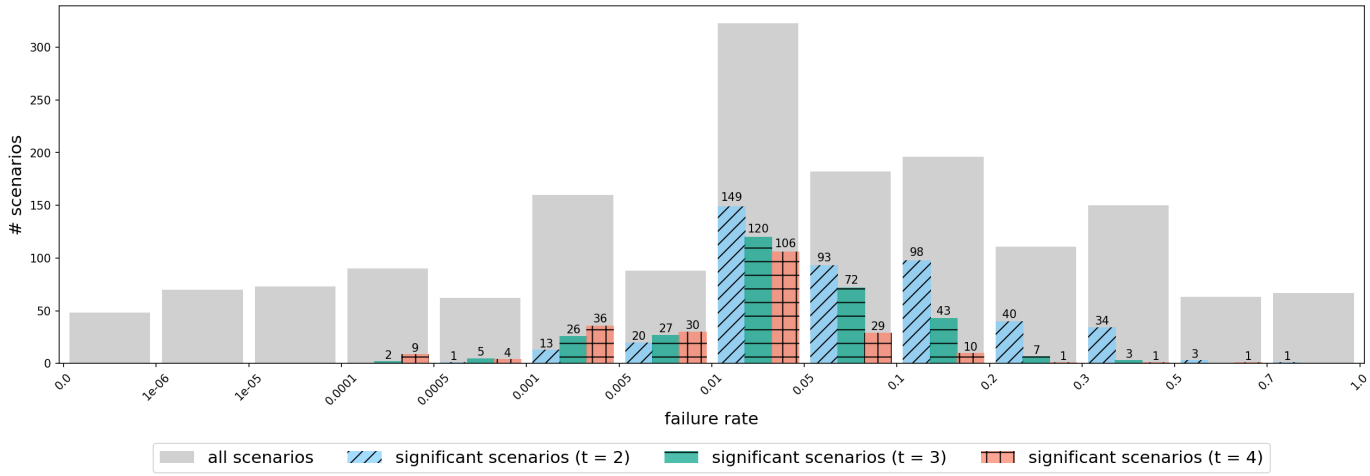

Fig. 5. Distribution of significant scenarios under different intervals of $rate$.

Figure 4 shows that the number of all scenarios increases with increase in $para$. This is because some faults can only be detected when a large number of parameters are involved, so more test scenarios were created when $para$ takes larger values. The smallest number of significant scenarios occurs for $cons = 0$, where only 4% (44/1044) of scenarios are significant. This indicates that the three techniques tend to be indistinguishable when constraints are unavailable in the model for a highly-constrained input space. Compared with it, more significant scenarios are observed for $cons > 0$. In particular, when $para = 1.0$ and $cons = 0.75$, we can see the highest proportion of significant scenarios (50%) for covering strength 2. This indicates that differences between the three techniques will have higher chances of occurring for models with partial constraints. This scenario is likely to occur in practice, since modelling configurable systems is a non-trivial task, especially if there are many dependencies between software configurations.

Figure 5 shows the distribution of significant scenarios under different intervals of $rate$. As $rate$ can take a series of continuous values and they are not uniformly distributed (see Figure 1), we divided the range $[0, 1]$ into uneven intervals and calculated the number of both significant and all scenarios that fall into each interval. For example, there are 160 scenarios whose $rate$ is in $[0.001, 0.005]$, and the number of significant scenarios is 13, 26 and 36 for each covering strength.

Figure 5 shows that with increase of $rate$ from 0 to 1, the number of significant scenarios increases first and then decreases. This suggests that performance differences between the three testing techniques are more likely to occur when $rate$ takes relatively low values. Specifically, on average 29% of scenarios are significant when $rate$ lies in $[0.001, 0.2]$. Whereas this proportion is only 4% for the other intervals. Additionally, the performance differences diminish to insignificant level when $rate$ is either very low or very high. Among all scenarios of the subject programs, the lowest and highest $rate$ investigated are as 7×10^{-11} and 1.0. By contrast, the range of $rate$ of all significant scenarios is $[0.0001, 0.71]$ for all covering strengths. Overall, the average value of $rate$ in all significant scenarios is 0.08.

TABLE 5
The number of scenarios where technique A is significantly superior ($+$), indistinguishable ($=$), or significantly inferior ($-$) to technique B.

τ	CT / RT	CT / ART	RT / ART
	$+$ / $=$ / $-$	$+$ / $=$ / $-$	$+$ / $=$ / $-$
2	377 / 1280 / 26	93 / 1548 / 42	1 / 1417 / 265
3	264 / 1408 / 11	36 / 1633 / 14	5 / 1469 / 209
4	195 / 1481 / 7	24 / 1649 / 10	5 / 1511 / 167

The three testing techniques tend to perform differently when the fault is relatively hard-to-detect.

Summing up, we answer $\mathbf{RQ_1}$ as follows: the effectiveness of CT, RT and ART tends to be significantly different when the failure rate is relatively low (between 0.001 and 0.2), and we did not observe any difference when the failure rate is lower than 0.0001 or higher than 0.71. The three testing techniques also tend to perform differently when a small test suite is generated, i.e., of the size of a pairwise test suite. But if the test model of a highly-constrained program contains no constraints ($cons = 0.0$), the three techniques tend to become indistinguishable in terms of fault detection.

4.2 $\mathbf{RQ_2}$: Favourable techniques

The second research question asks which techniques perform best under different test scenarios. To answer this question, we compared fault detection ability between each pair of techniques, namely CT/RT, CT/ART and RT/ART.

For each scenario, Tukey's HSD test was used to determine whether there is a significant difference in performance for each pair of techniques. Table 5 shows the number of scenarios where testing technique A is significantly superior ($+$), indistinguishable ($=$), or significantly inferior ($-$) to testing technique B across all subject programs for each covering strength. From Figure 5, we can see that any pair of the techniques investigated is indistinguishable in more than 1200 out of 1683 scenarios for each covering strength. This is due to the fact that the significant difference of CT, RT and ART is only observed in 19% of scenarios (the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

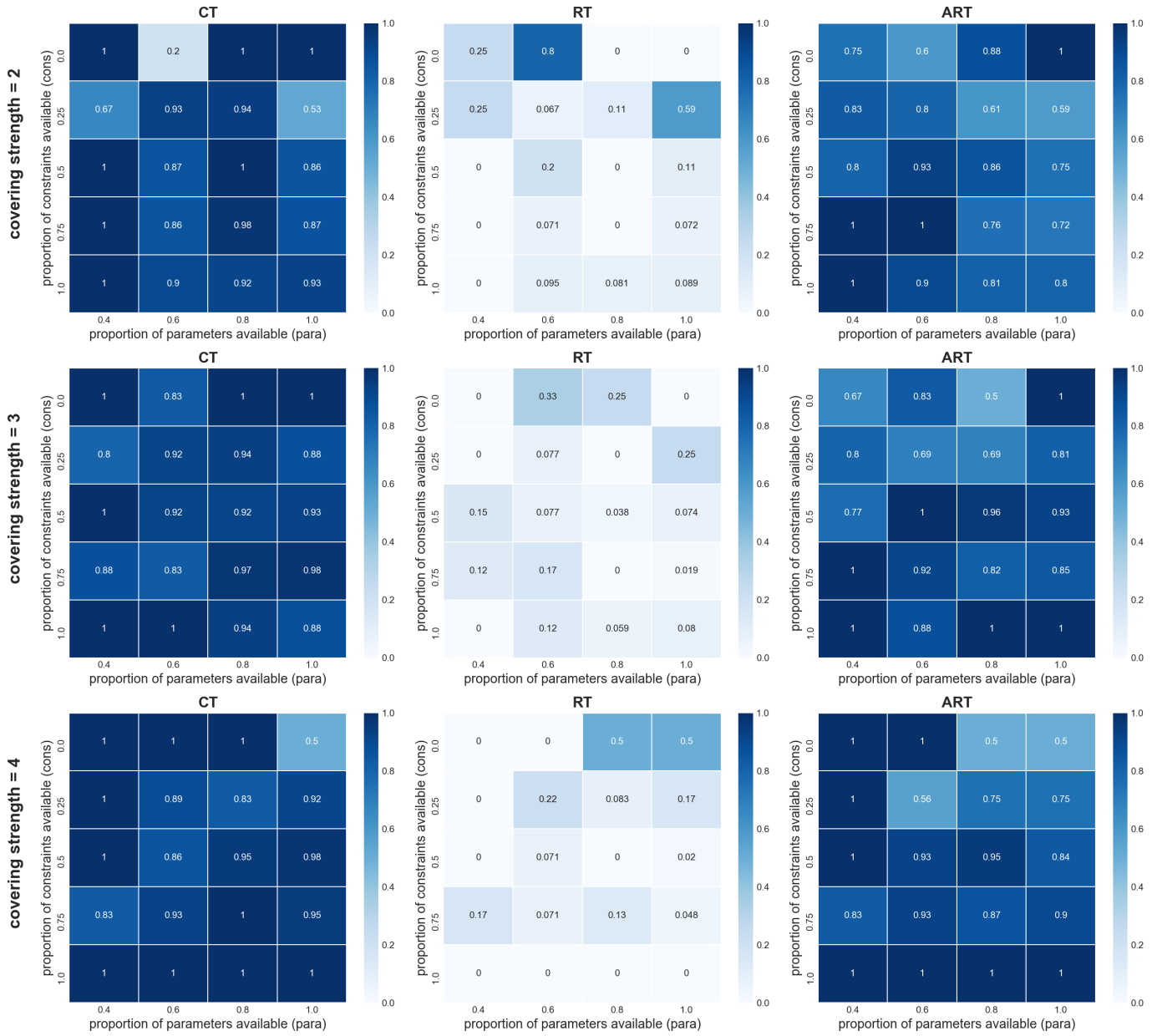

Fig. 6. Proportion of significant scenarios in which each testing technique is favourable under different values of $para$ and $cons$.

characteristics of those significant scenarios that make the three techniques distinguishable are investigated in RQ_1; specifically, the three techniques tend to perform differently when the failure rate is relatively low, or a proportion of constraints is present in the test model). Regarding the relative performance between each pair of techniques, overall CT significantly outperforms RT in a total of 836 scenarios, but is worse in 44 scenarios. For ART, there are only 11 scenarios where RT significantly outperforms ART among all covering strengths. Otherwise ART is no worse than RT. Moreover, in 96% of scenarios CT and ART are indistinguishable. Especially for $\tau = 4$, the difference is only observed in 34 scenarios. This suggests that ART is usually as effective as CT.

Next, we determine which techniques are favourable for each of the significant scenarios. For a test scenario, a technique is 'favourable' if this technique is not significantly worse than the others, and so it can be recommended for testing. According to Tukey's HSD test, the techniques that have grouping letter 'a' are exactly the favourable techniques for each scenario, as they belong to the group of the best performance and they are not significantly different from each other. For example, assume a scenario where the grouping letters assigned to CT, RT and ART are a, b and ab, respectively. This means that CT is significantly better than RT, and ART is not distinguishable from either CT or RT. In this case, CT and ART are favourable techniques as both of them are not worse than the other technique, i.e. RT.

Figure 6 shows the proportion of significant scenarios in which each testing technique is favourable, under different values of $para$ and $cons$ for $\tau = 2, 3, 4$. Each two dimensional coordinate in each figure denotes a particular

TABLE 6
Proportion of significant scenarios in which each testing technique is favourable under different intervals of $rate$.

interval	a	0	10^{-6}	10^{-5}	0.0001	0.0005	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5	0.7
$(a, b]$	b	10^{-6}	10^{-5}	0.0001	0.0005	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5	0.7	1.0
$\tau = 2$	CT	–	–	–	–	1.00	0.85	0.80	0.92	0.90	0.78	1.00	1.00	1.00	1.00
	RT	–	–	–	–	0.00	0.23	0.20	0.10	0.10	0.14	0.03	0.00	0.00	0.00
	ART	–	–	–	–	1.00	0.62	0.60	0.69	0.80	0.86	0.95	1.00	1.00	1.00
$\tau = 3$	CT	–	–	–	1.00	0.60	0.88	0.89	0.93	0.97	0.95	1.00	1.00	–	–
	RT	–	–	–	0.50	0.20	0.12	0.15	0.07	0.03	0.05	0.00	0.00	–	–
	ART	–	–	–	0.00	0.80	0.85	0.74	0.82	0.94	1.00	1.00	1.00	–	–
$\tau = 4$	CT	–	–	–	0.78	1.00	0.92	1.00	0.93	1.00	1.00	1.00	1.00	1.00	–
	RT	–	–	–	0.33	0.00	0.11	0.03	0.06	0.00	0.00	0.00	0.00	0.00	–
	ART	–	–	–	0.44	0.50	0.75	0.77	0.95	1.00	1.00	1.00	1.00	1.00	–

combination of $para$ and $cons$; a darker colour indicates a higher proportion. For example, in Figure 6, for $\tau = 2$ and $para = 1.0$ and $cons = 1.0$, there are 56 significant scenarios, and CT, RT and ART are favourable in 93%, 8.9% and 80% of them, respectively (in these scenarios, they are significantly better than at least one of the other techniques).

Figure 6 shows that the proportion of significant scenarios favoured by CT is not lower than those of RT and ART in 48 out of 60 combinations of $para$ and $cons$ (for all covering strengths). Specifically, among 984 significant scenarios, CT is identified as favourable in 903 (92%) cases, where CT is at least statistically significantly better than one of the other two techniques. If we further consider the scenarios where the three techniques perform equally well, overall CT is no worse than the others in 98% of cases. This finding demonstrates that CT is the most favourable among the three techniques. In addition, when $cons \geq 0.5$, CT is favourable in at least 83% of significant scenarios. CT is unlikely to perform worse when a relatively high proportion of constraints are present in the model.

However, there are 81 scenarios where CT is significantly worse than RT or ART in terms of fault detection across all programs; 43% of these scenarios appear in the case of FLEX. For these 81 scenarios, we observe that only 13 (16%) of them have presented all constraints in the model ($cons = 1.0$). Note that invalid combinations will prevent the test cases containing them from execution. If a test case covers both a failure causing combination and an invalid combination, the fault cannot be detected as the failure causing combination is masked by constraint violation (a result of masking effect [57]). Consequently, except covering failure causing combinations, in some cases some other parameters also need to take fixed values to make the test case valid for triggering the fault. When the total number of parameters that need to be fixed exceeds the covering strength τ, CT might not generate a test case that would trigger such a fault, especially when $\tau = 2$ is applied.

As far as ART is concerned, ART is favourable not only in 58 out of 82 significant scenarios where RT is favourable, but also in much more scenarios where RT is not favourable. This demonstrates the superiority of ART to RT, regardless of the test scenarios. Moreover, among all significant scenarios where CT is determined as the favourable technique, ART is also recommended in 83% of them. This further suggests the effectiveness of ART: it can perform as well as CT to some extent. Furthermore, there are also 77 scenarios (8% of significant scenarios) where ART is favourable but CT is not. In these cases ART performs better than CT.

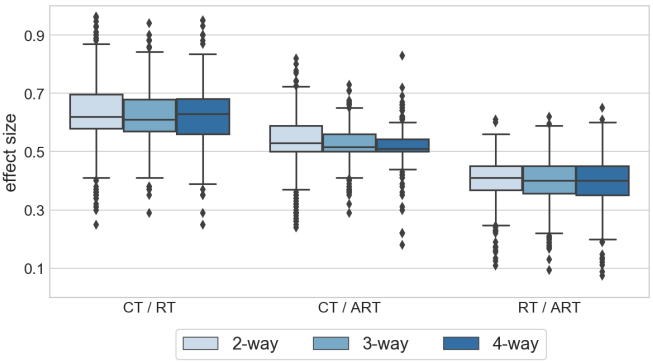

Fig. 7. The distribution of $\widehat{A}_{12}$ statistics obtained from significant scenarios for $\tau = 2, 3, 4$.

Table 6 shows the proportion of significant scenarios in which each testing technique is favourable, under different intervals of $rate$ for $\tau = 2, 3, 4$. The first two lines indicate the interval (left-open, right-closed), and '$-$' indicates that there are no significant scenarios. For example, for $\tau = 2$ and $rate \in (0.001, 0.005]$, CT, RT and ART are favourable in 85%, 23% and 62% of significant scenarios, respectively.

From Table 6, we have similar conclusions to Figure 6: CT is generally the most favourable technique, and ART demonstrates relatively similar behaviour to CT. Moreover, for $rate \leq 0.005$, on average, CT is favourable in 88% of significant scenarios, which is noticeably higher than those of RT (19%) and ART (62%). This suggests that CT is more desirable when the fault is very hard to detect.

Additionally, to quantify the size of the differences for each pair of the three testing techniques, we applied Vargha and Delaneys $\widehat{A}_{12}$ statistic [56] on CT/RT, CT/ART and RT/ART for each significant scenario. Figure 7 shows the distribution of $\widehat{A}_{12}$ obtained for each covering strength. A higher $\widehat{A}_{12}$ indicates that the first technique has a higher probability of outperforming the other.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

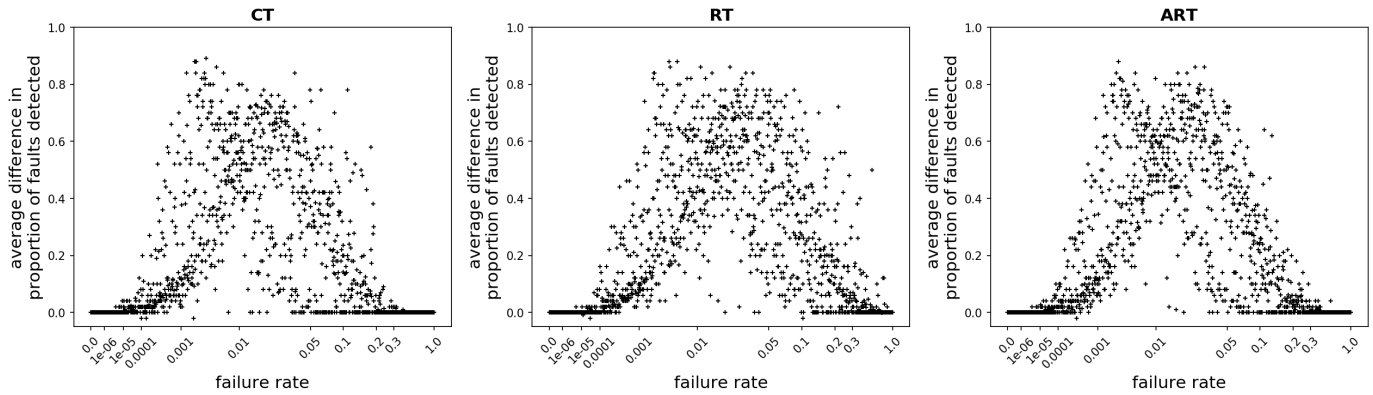

Fig. 8. The average difference in fault detection ability between 4-way and 2-way CT, RT and ART for different values of $rate$.

TABLE 7
The average difference in fault detection ability between 4-way and 2-way CT, RT and ART for different values of $para$ and $cons$.

			proportion of constraints ($cons$)				
			0.0	0.25	0.5	0.75	1.0
CT	proportion of parameters ($para$)	0.4	0.22	0.31	0.18	0.16	0.07
		0.6	0.29	0.22	0.28	0.20	0.06
		0.8	0.19	0.23	0.30	0.25	0.05
		1.0	0.16	0.21	0.36	0.28	0.06
RT	proportion of parameters ($para$)	0.4	0.22	0.26	0.21	0.19	0.12
		0.6	0.26	0.26	0.29	0.22	0.09
		0.8	0.21	0.26	0.33	0.35	0.14
		1.0	0.16	0.20	0.34	0.40	0.16
ART	proportion of parameters ($para$)	0.4	0.21	0.26	0.20	0.17	0.08
		0.6	0.27	0.24	0.30	0.21	0.06
		0.8	0.21	0.27	0.31	0.29	0.07
		1.0	0.15	0.20	0.37	0.32	0.09

From Figure 7, when comparing CT and RT for $\tau = 2$, we observe that the effect size is greater than 0.5 (CT is better than RT) for more than 75% of significant scenarios. But there are also 4% of significant scenarios where CT has at most 40% chance of outperforming RT. Regarding CT and ART, the median of effect sizes is very close to 0.5 for all programs. Especially for $\tau = 4$, the median is equal to 0.5, which confirms the indistinguishable performance between CT and ART. In addition, when comparing RT and ART, for more than 75% of significant scenarios the effect size is less than 0.5. This confirms the superiority of ART to RT.

Summing up, we answer $\mathbf{RQ_2}$ as follows: CT is no worse than RT and ART in 98% of test scenarios, especially when the failure rate is low ($rate \leq 0.005$), making it the most favourable out of the three techniques investigated. CT is also desirable when all constraints are present in the model ($cons = 1.0$), as 84% of scenarios where CT is worse than the others have $cons < 1.0$. As an improvement of RT, ART enhances RT in almost all scenarios. Moreover, ART is indistinguishable from CT in 96% of test scenarios, especially when a large test suite ($\tau = 4$) is used.

4.3 $\mathbf{RQ_3}$: Covering Strength

The third research question asks how the three testing techniques perform depending on the covering strength of the test suite used. In Section 4.1 and 4.2, we showed that the three techniques tend to be indistinguishable when a high covering strength (large test suite) is applied, and CT is generally the best for all covering strengths. The remaining question is how much difference exists between lower and higher covering strengths for different scenarios. For example, if 2-way and 4-way testing detect similar number of faults for a scenario, then applying 2-way CT first might be recommended in practice.

To investigate the impact of covering strength on the effectiveness of the three techniques, for each scenario we evaluated the difference between 4-way and 2-way testing for each of CT, RT and ART in terms of:

$$\begin{aligned} difference = &\text{ proportion of faults detected by 4-CT/RT/ART} \\ &- \text{ proportion of faults detected by 2-CT/RT/ART} \end{aligned}$$

where τ-RT and τ-ART are used to represent that RT and ART are applied to generate test suites with the same size as τ-CT. We only considered $\tau = 4$ and 2 as they represent the potential best and worst performance for the considered techniques.

Table 7 shows the average difference obtained for different values of $para$ and $cons$. We observe that the differences are all greater than 0, indicating that 4-way testing improves the fault detection ability in almost all cases.

For different choices of $para$ and $cons$, from Table 7 we observe that, as $cons$ increases, the improvement of 4-way testing tends to first slightly increase, before subsequently decrease, for all choices of $para$ and all testing techniques. Specifically, the smallest improvement of 4-way testing occurs when all constraints are present in the model ($cons = 1.0$), where on average the differences are only 0.06, 0.13 and 0.08 for CT, RT and ART, respectively. By contrast, the improvement of 4-way testing is at least 0.15 with 0.25 average for $cons < 1.0$. This finding indicates that a large test suite is more beneficial for detecting faults when only partial constraints are available in the model. When all constraints are identified, we can apply $\tau = 2$ at first and increase covering strength in an incremental manner.

Figure 8 shows the average difference in proportion of faults detected for different values of $rate$. Note that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2852744, IEEE Transactions on Software Engineering

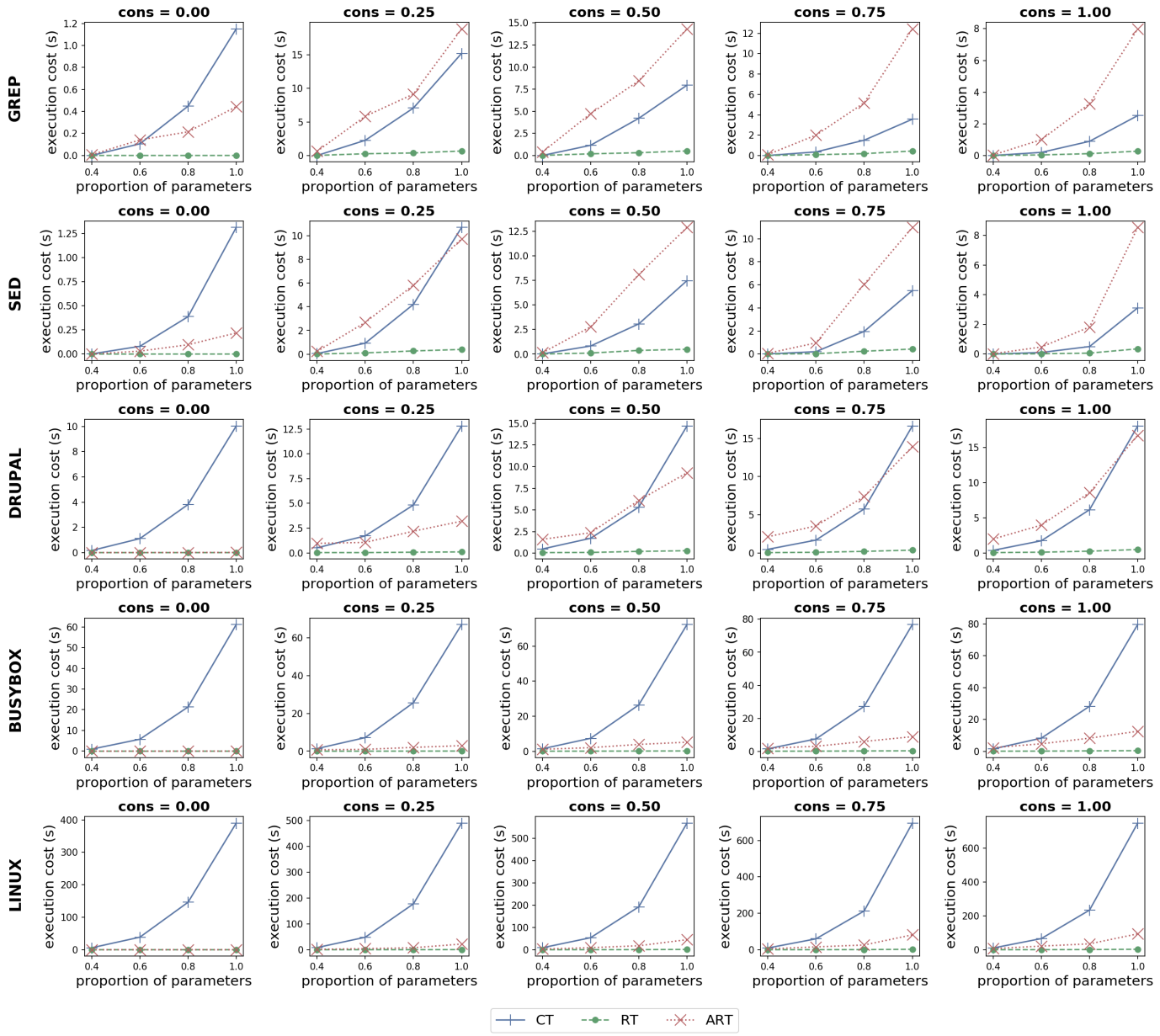

Fig. 9. Computational cost of CT, RT and ART for GREP, SED, DRUPAL, BUSYBOX and LINUX for $\tau = 4$.

the x-axis is adjusted due to the uneven distribution of failure rates. From Figure 8, we can see that the difference in performance between 4-way and 2-way testing always increases first and then decreases with increase in $rate$ (we attribute the slight multimodal distribution to different subject programs[6]). The largest improvement occurs when the failure rate is relatively low (between 0.001 and 0.05), where 4-way testing can detect up to 89% more faults than 2-way testing. When the failure rate exceeds 0.3, the difference tends to diminish, and so applying $\tau = 2$ is good enough to trigger these easy-to-detect faults. The difference also tends to diminish at failure rates below 1×10^{-5}, where even 4-way testing is not likely to trigger these almost-impossible-to-find faults.

Summing up, we answer $\mathbf{RQ_3}$ as follows: in all scenarios a larger test suite generally leads to a better performance. When only partial constraints are available in the model ($cons < 1.0$), the average difference in proportion of faults detected between 4-way and 2-way testing is 0.25, which makes higher covering strength desirable for these scenarios. Higher covering strength is much more desirable when the failure rate is relatively low (between 0.001 and 0.05), where up to 89% of faults can be further detected. But when failure rate exceeds 0.3, higher covering strength provides few benefits.

6. For GREP and SED, the largest improvement occurs when $rate$ is in $(0.001, 0.01)$; for the others, it occurs when $rate$ is in $(0.01, 0.05)$. The more details can be found on this paper's companion website: http://gist.nju.edu.cn/doc/ec18/.

4.4 $\mathbf{RQ_4}$: Efficiency

The last research question asks about the computational cost of CT, RT and ART under different test scenarios. Given a

scenario, the time duration of test suite generation is only related to parameters and constraints. So we only consider the impact of $para$ and $cons$ on the computational cost of the three techniques. Additionally, due to the greedy nature of the one-test-at-a-time framework that is applied to generate test suites, in most scenarios all three techniques can finish test suite generation very quickly (within 6 seconds for each scenario in our case). Therefore, we only consider the case with the highest computational cost, i.e. $\tau = 4$.

The total time costs of CT, RT and ART across all test scenarios from all programs for covering strength 4 are 4910, 24 and 742 seconds, respectively. Generally CT is the most expensive with ART being more expensive than RT in terms of computational cost. This conforms with conventional wisdom that ART requires more time than RT in order to calculate the distance between new and previous test cases at each iteration, while CT requires even more time to calculate the number of covered combinations.

Figure 9 shows the computational cost of each technique for GREP, SED, DRUPAL, BUSYBOX and LINUX for different values of $para$ and $cons$. We only considered these five programs as they have the largest search spaces among our subjects as shown in Table 4. For other programs, the observed time costs are all less than 6 seconds.

From Figure 9, we conclude that the computational cost of RT is negligible in almost all cases. The computational cost of ART and CT increases with increase in $para$, as the more parameters are present in the model, the larger is the search space. When no constraints are available ($cons = 0$), CT requires much more time than ART, where the time cost of ART is no more than one second, but the time cost of CT can be up to 390 seconds for the model with 104 parameters (i.e., LINUX). However, when constraints are known to the tester ($cons \geq 0.25$), the time cost of ART might increase significantly. In particular, for GREP and SED, ART tends to spend more time than CT; especially when $cons \geq 0.75$, the time cost of ART might be up to 3.5 times higher than that of CT.

Note that for GREP and SED, every parameter is involved in at least one constraint, which makes them the most highly-constrained programs. While for DRUPAL, BUSYBOX and LINUX, the proportion of parameters involved in constraints are 79%, 38% and 66%, respectively. To generate the next test case, the ART algorithm used in this work needs to randomly sample 30 valid test cases at first and then select the best one from them. If a program is highly constrained, a large number of constraints will be present in the model. As a result, a randomly sampled test case is less likely to be valid, and so a higher computational cost of ART is observed for GREP and SED in Figure 9. Whereas, when there is a large number of parameters, such as in BUSYBOX and LINUX, CT still tends to spend more time than ART.

Summing up, we answer $\mathbf{RQ_4}$ as follows: RT is always the fastest technique in all scenarios. When no constraints are present in the model ($cons = 0$), ART is much faster than CT as it spends no more than one second. ART is also faster than CT when there is a large number of parameters. However, when there is a large number of constraints present in the model and all parameters are involved in constraints, ART tends to spend more time than CT. In particular, ART is up to 3.5 times slower than CT on our test subjects.

5 Discussion

We found that in 98% of test scenarios combinatorial testing (CT) is no worse in terms of fault detection ability than random testing (RT) and adaptive random testing (ART). Therefore, when effectiveness is the key priority, we recommend using CT. Moreover, the three testing techniques tend to be significantly distinguishable when the failure rate is relatively low. In this case, higher-strength CT is helpful for detecting those hard-to-detect faults. CT is also likely to perform better when all constraints are correctly identified, as 84% of scenarios where CT is significantly worse than RT or ART are with partial constraints in test models.

However, deriving CT test suites will spend a lot of time. Our results on scalability of CT re-confirm earlier findings [21] that also found that CT does not scale to the size of the original model of LINUX (with more than 30,000 parameters) for strengths above 2. Hence further research is needed to improve the scalability of CT for real-world large programs.

In addition, the time cost of CT comes from not only running the CT generation algorithm, but also from deriving a complete CT model. Given a highly-constrained program, the three testing techniques tend to exhibit similar behaviour in more than 90% of scenarios with no constraints available in the test model. In this case, RT is more preferable due to its low computational cost. Furthermore, as RT can be as effective as CT in test scenarios where the failure rate is high, it could be used to efficiently discover the easy-to-detect faults, such as those likely to be present in a poorly implemented software.

ART is a similarity based test suite generation technique. The effectiveness of similarity heuristic in terms of combination coverage has been demonstrated in [27]. We found that in 96% of test scenarios ART performs as well as CT in terms of fault detection ability, suggesting that ART can be indeed used as an alternative to CT to some extent. Moreover, for large test models such as BUSYBOX and LINUX, ART is more computational cost-effective, because it is usually faster than CT. However, when the model is highly constrained and a high proportion of constraints are present in the model, we found that the computational cost of ART can be as much as 3.5 times than that of CT, which makes ART less desirable. As in this work we only applied a simple constraint handling strategy for FSCS-ART algorithm, further improvements are needed on the constrained ART test suite generation.

As far as the covering strength of the test suite is concerned, the difference between the three testing techniques tends to diminish when a higher τ (larger test suite) is used. This suggests that RT and ART could also perform well if there is less limitation on the testing resources. Additionally, performance differences between τ-CT, τ-RT and τ-ART are usually indistinguishable when constraint information is unavailable in the model ($cons = 0$). At the same time, a higher τ is desirable as the average difference in proportion of faults detected between 4-way and 2-way testing can be 0.21 for $cons = 0$. Therefore, if the tester cannot identify any constraints for a highly-constrained program, a large RT test suite could be as good as the more sophisticated techniques such as CT and ART. By contrast, if the tester has confidence

that all constraints are present in the model ($cons = 1.0$), that average difference between 4-way and 2-way testing is only 0.09. In this case, it might suggest to apply 2-CT or 2-ART at first, and then increase covering strength with an incremental manner.

Additionally, although RT and ART can be more favourable than CT in many test scenarios, one unparalleled advantage of τ-way CT is that it provides 100% τ-way combination coverage. If such a guarantee is desired, CT should be applied.

Overall, when the faults tend to be easy to detect (for example, in a poorly implemented software) or only a few constraints tend to be identified (for example, the tester has insufficient knowledge of the software), RT is preferable as it is usually more efficient and no less effective than CT and ART. By contrast, if the faults tend to be harder to detect and test model tends to be more complete, CT and ART will likely perform better. However, if the test model is highly constrained (all parameters are involved in the constraints), CT is preferable as current ART's constraint-handling becomes less effective.

6 Threats to Validity

As far as internal threats to validity are concerned, the performance of CT, RT and ART depends on their particular implementations. It is possible that slightly different results will be observed by applying different generation algorithms. Although there are some widely used tools for covering array generation, there are no available tools to generate RT and ART test suites in the presence of constraints. To avoid potential bias, we applied the same one-test-at-a-time framework for all three testing techniques, and implemented all generation algorithms ourselves. All three techniques use the same framework with the only difference being the selection strategy for the next test case.

We acknowledge that there are state-of-the-art tools available for combinatorial testing. There has been a lot of work available comparing random and combinatorial testing. In order to avoid environmental bias and focus on the key subject that differentiates the different strategies, that is the test case selection criterion, we decided to focus on the one-test-at-a-time framework. Furthermore, this allows for all test suites to be generated in a reasonable time, especially for the three larger test models. Since CT turned out to be the superior out of the three in most test scenarios, we predict that the results would be even stronger for CT test suites generated using state-of-the-art CT algorithms. These would produce smaller test suites, thus RT and ART test suites of the same size would have even less chance of detecting faults. One of the state-of-the-art one-test-at-a-time CT algorithms is the AETG algorithm [35]. We implemented the most well-known version of it. We used the most popular version of ART as argued in the survey [7]. We acknowledge that more sophisticated CT and ART variants have been developed [22], [41], which may produce smaller test suites of CT, or more diverse test suites of ART.

Another internal threat to validity is the modelling of subject programs. The creation of the 'perfect' test model (including the identification of parameters, values and constraints) is a challenging task. Indeed, it is an open problem in CT [1]. In this study, we used the same test models, or extracted subsets of the test models, from previous work [21], [33], to avoid potential bias for choosing models for these programs. We acknowledge that different testers might create different test models, such as identifying different parameters and constraints, encoding constraints into different formats, and assigning priorities to different constraints. In addition, we used an existing corpus of faults (manually seeded or mined from the repository) in this study. This ensures to include some real faults (reported and confirmed by the original developers), and avoids biasing the fault detection to any particular testing technique. We acknowledge that CT, RT and ART might discover other faults in those subject programs.

As far as external threats to validity are concerned, the main threat to this work is that we conducted experiments on only nine subject programs. As a result, different test scenarios, such as different bounds of failure rates, may be determined and recommended for the three testing techniques. The six SIR programs are relatively small (with respect to the number of parameters), but they are from a well-studied software repository and have been widely used in the CT literature [29], [47], [48], [49], [50]. We also used three larger highly-configurable programs (larger test models) with real faults [21], [33]. These nine programs vary in both types and sizes. We believe that they represent realistic scenarios for comparing the three testing techniques.

7 Related Work

In order to evaluate the effectiveness of CT, comparisons between CT and RT have been made. CT has also been compared with manually generated test cases and other test techniques. However, conflicting results are reported in the literature.

In general, CT is expected to detect more faults than RT, or require less test cases to achieve the same combination coverage. Dunietz et al. [8] evaluated τ-way and random testing in terms of code coverage on an operations support system. They found that CT with low covering strength could be effective for block coverage. Bell and Vouk [10] compared 2-way and random testing for detecting security faults on two network related products, and Pretschner et al. [11] compared them for testing access control policies on four cases. Both studies showed the superiority of 2-way CT. Additionally, Kobayashi et al. [9] applied τ-way and random testing to examine logical expressions from the specification of TCAS II. Their work was further expanded by Ballance et al. [17] and Vilkomir et al. [18], where automatically generated expressions were used for simulation. These three investigations all found that CT has an advantage over RT. Moreover, Calvagna et al. [12] compared CT and RT for conformance testing on the byte code verifier component of the Java virtual machine. Their results not only demonstrated the superiority of CT, but also showed that CT is able to detect a wider set of faults that cannot be detected by random test suites of comparable or even larger size. Recently, Medeiros et al. [21] conducted a large empirical study to compare 10 sampling algorithms on 24 open-sources configurable software systems. They found that CT with higher covering strength can detect more

faults, but regarding the tradeoffs between test suite size and fault detection ability, the simple most-enabled-disabled algorithm, i.e. 1-way CT, is the most efficient one.

Although it has been shown that CT outperforms RT, some opposite findings were also reported. Dalal and Mallows [19] compared the combination coverage of several optimal covering arrays with same sized random test suites. They observed that RT can cover as high as 90% τ-way combinations. Schroeder et al. [13] evaluated τ-way and random testing on two industrial programs with manually seeded faults. They found that there is no significant difference between these two techniques. Bryce et al. [14] compared the structural and combination coverage of τ-way and random test suites on a component of the Flight Guidance System. They found that CT provides little benefit over RT and does not improve requirement-based testing. Additionally, Ghandehari et al. [20] compared code coverage and fault detection ability of CT and RT on the Siemens Suite. They concluded that the differences between these two techniques are not as significant as people would have probably expected.

Moreover, some comparisons of CT and RT revealed that their relative effectiveness may be dependent on the covering strength. Ellims et al. [15] compared τ-way, random and manuallly generated test suites on a system that controls a large industrial engine. They found that 2-way CT is not adequate, but CT with higher covering strength could perform at least as well as a manually generated test suite. Kuhn et al. [16] compared τ-way and random testing for detecting deadlocks on a network simulator. They found that 2-way CT detects slightly fewer deadlocks than a random test suite of the same size, but 4-way CT performs better than RT.

Additionally to empirical studies, Arcuri and Briand [5] have made a formal analysis on the probability of detecting interaction triggered faults of CT and RT. Their results showed that a random test suite of the same size as a τ-way covering array could trigger at least one τ-way fault with a probability of more than 63%. Furthermore, they conclude that with increased number of parameters, RT could be more effective and will converge towards equal effectiveness as CT. However, their analysis assumes there are no constraints in the system under test.

On the other hand, as ART is originally designed to enhance the fault detection ability of RT, there are lots of studies to compare these two techniques and almost all of them have consistently demonstrated that ART outperforms RT. For example, simulated result was reported in [22], while empirical studies were conducted under various open source programs with both seeded faults [23], [25], [26] and real-life faults [24]. However, these studies did not consider the impact of constraints.

The goal of ART is to achieve a diverse test suite. Such process will make test cases as different as possible, which also implies that these test cases could cover different τ-way combinations and thus a relatively high combination coverage can be achieved. With this idea in mind, Henard et al. [27] have proposed a search based technique that employs similarity-based heuristic to generate and prioritise configurations for software product line (SPL). Their experiment showed that although this technique does not guarantee to achieve full combination coverage, it can scale to large SPLs in a reasonable time and still cover a large portion of τ-way combinations. This finding suggests that similarity-based test generation, or ART, may be used as a viable alternative to τ-way testing. However, only combination coverage was evaluated in this study.

TABLE 8
A detailed comparison between our previous study and this one.

	The previous study [28]	This study
SUT	Unconstrained.	Constrained.
Test Model	The tester can determine all relevant parameters and constraints.	The tester may be unaware of all relevant parameters and constraints.
Fault	Synthetic. Each fault is caused by one (and only one) k-way combination.	Real/Seeded. A fault may be caused by multiple combinations.
Experiment	Simulation (synthetic models).	Empirical study (real-world programs).
Focus	Ability of τ-way CT, RT and ART of hitting k-way faults, when $k > \tau$, $k = \tau$ and $k < \tau$.	Performance of CT, RT and ART under realistic testing scenarios, with incomplete models and varying failure rates*.
Summary	Idealised.	More practical.

$\star$ Failure rates are related to how many test cases can trigger a fault (and thus how easily detectable the faults are) and not how many parameter settings can trigger a fault as in [28].

Although many comparisons of CT and RT, and of ART and RT, have been made, the only study that compares CT, RT and ART at the same time is from Nie et al. [28]. In that work, the three techniques were compared in terms of the ability of detecting interaction triggered faults. The results showed that in detecting k-way faults with $k \leq \tau$, τ-way CT is better than RT and ART when using the same-sized test suites. However, when detecting k-way faults with $k > \tau$, τ-way CT has no advantages. These findings explain in part why some studies observe no significant difference between RT and CT.

Note that although the previous study [28] and this one both compare CT, RT and ART, they are different in many aspects, as summarised in Table 8. The previous study [28] focuses on the impact of the number of parameters that are involved in the fault, but in this study, our focus is the completeness of test models (including parameters and constraints) and the difficulty of fault detection. Moreover, the previous study [28] is based on some ideal assumptions: the SUT is unconstrained and every fault is caused by an exact k-way combination, thus only simulated experiment was conducted (in a rather idealised scenario). Instead, as most programs have constrained domains and software faults may be caused by multiple combinations [21], [29], [30], this study can provide more practical suggestions based on more realistic test scenarios.

Summing up, Table 1 in Section 1 has given an overview of literature on comparisons among CT, RT and ART. Although many studies have been made, there still exist some shortcomings: most comparisons of CT and RT are based on case analysis, where only two studies reported empirical

findings [20], [21]; an empirical comparison of CT and ART is not available; many previous studies only considered unconstrained test models, especially when comparing ART with RT [22], [23], [24], [25], [26].

Moreover, previous studies all assumed the usage of a complete test model, in which all relevant parameters and constraints are known to the tester. However, as modelling remains a difficult task with no automated tools that can be used [1], the tester may not be able to be aware of all parameters and constraints and, therefore, be unable to avoid omitting them. Researchers also proposed adaptive strategies to relieve testers of the need to make a perfect test model before testing [2], [3]. In addition, different kinds of faults, as characterised by their failure rate, may pose different challenges for different testing techniques. Hence, there is a need to empirically compare CT, RT and ART and report observations with respect to different test scenarios (parameters, constraints and failure rates) in order to achieve a better understanding of their relationships.

8 Conclusions

We empirically compared three popular testing techniques, namely, CT, RT and ART, under different test scenarios. Their fault detection ability and computational cost were evaluated under different choices of the proportion of parameters available ($para$) and the proportion of constraints available ($cons$) in the model, and the fault failure rates ($rate$). Our experiment was conducted on nine real-world programs with real faults, and a total of 1683 different test scenarios were used for evaluation.

Our results show that there tends to be a significant difference in the fault detection ability of the three testing techniques when the failure rate is relatively low. However, when a large test suite is used, or no constraints are present in the model for a highly-constrained program, the difference in performance tends to diminish to an indistinguishable level. In general, CT is recommended to be applied as it performs no worse than the other two techniques in 98% of test scenarios studied, and it becomes more favourable when the fault is hard to detect and all constraints are correctly identified in the model. As an improvement of RT, ART does enhance RT, and is indistinguishable from CT in 96% of test scenarios. However, when there is a large number of constraints present in the model and all parameters are involved in constraints, ART might be significantly less efficient than RT and CT.

Overall, the relative performance of the three testing techniques is dependent on the interplay between fault revealability and the testers' knowledge of the parameters and constraints. RT is preferable when the faults tend to be easy to detect and fewer constraints tend to be available in the test model. However, when the faults become harder to detect and test models become more complete, we recommend using CT or ART, with CT being preferable for highly-constrained programs.

The test models and corpus of faults used in this study, together with our codes, experimental results and all plots with respect to each subject program are available on this paper's companion website: http://gist.nju.edu.cn/doc/ec18/.

Acknowledgments

This work was partially supported by the National Key Research and Development Plan (No. 2018YFB1003800) and the Program B for Outstanding PhD Candidate of Nanjing University (No. 201701B028).

References

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[2] C. Nie, H. Leung, and K.-Y. Cai, "Adaptive combinatorial testing," in *International Conference on Quality Software*. IEEE, 2013, pp. 284–287.

[3] C. Yilmaz, S. Fouche, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc, "Moving forward with combinatorial interaction testing," *Computer*, vol. 47, no. 2, pp. 37–45, 2014.

[4] A. Arcuri, M. Z. Iqbal, and L. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 219–230.

[5] A. Arcuri and L. Briand, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1088–1099, 2012.

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[8] I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. L. Mallows, and A. Iannino, "Applying design of experiments to software testing: experience report," in *Proceedings of the 19th international conference on Software engineering*. ACM, 1997, pp. 205–215.

[9] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "Applicability of non-specification-based approaches to logic testing for software," in *International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 337–346.

[10] K. Z. Bell and M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software," in *International Conference on Information and Communications Technology*. IEEE, 2005, pp. 221–235.

[11] A. Pretschner, T. Mouelhi, and Y. Le Traon, "Model-based tests for access control policies," in *International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 338–347.

[12] A. Calvagna, A. Fornaia, and E. Tramontana, "Random versus combinatorial effectiveness in software conformance testing: a case study," in *Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1797–1802.

[13] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," in *International Symposium on Empirical Software Engineering*. IEEE, 2004, pp. 49–59.

[14] R. C. Bryce, A. Rajan, and M. P. Heimdahl, "Interaction testing in model-based development: Effect on model-coverage," in *Asia Pacific Software Engineering Conference*. IEEE, 2006, pp. 259–268.

[15] M. Ellims, D. Ince, and M. Petre, "The effectiveness of t-way test data generation," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2008, pp. 16–29.

[16] D. R. Kuhn, R. Kacker, and Y. Lei, "Random vs. combinatorial methods for discrete event simulation of a grid computer network," in *Proceedings of ModSim World*, 2009, pp. 83–88.

[17] W. A. Ballance, S. Vilkomir, and W. Jenkins, "Effectiveness of pairwise testing for software with boolean inputs," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 580–586.

[18] S. Vilkomir, O. Starov, and R. Bhambroo, "Evaluation of t-wise approach for testing logical expressions in software," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 249–256.

[19] S. R. Dalal and C. L. Mallows, "Factor-covering designs for testing software," *Technometrics*, vol. 40, no. 3, pp. 234–243, 1998.

[20] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn, "An empirical comparison of combinatorial and random testing," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 68–77.

[21] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *International Conference on Software Engineering*. ACM, 2016, pp. 643–654.

[22] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Transactions on Reliability*, vol. 58, no. 4, pp. 619–633, 2009.

[23] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[24] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: adaptive random testing for object-oriented software," in *International conference on Software engineering*. ACM, 2008, pp. 71–80.

[25] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of java programs," in *International Conference on Automated Software Engineering*. IEEE, 2009, pp. 221–232.

[26] B. Zhou, H. Okamura, and T. Dohi, "Enhancing performance of random testing through markov chain monte carlo methods," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 186–192, 2013.

[27] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[28] C. Nie, H. Wu, X. Niu, F.-C. Kuo, H. Leung, and C. J. Colbourn, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures," *Information and Software Technology*, vol. 62, pp. 198–213, 2015.

[29] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 901–924, 2015.

[30] J. Petke, "Constraints: the future of combinatorial interaction testing," in *International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2015, pp. 17–18.

[31] D. R. Kuhn and D. R. Wallace, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[32] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[33] A. B. Sánchez, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, "Variability testing in the wild: the drupal case study," *Software & Systems Modeling*, vol. 16, no. 1, pp. 173–194, 2017.

[34] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[35] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[36] C. J. Colbourn, "Covering arrays, augmentation, and quilting arrays," *Discrete Mathematics, Algorithms and Applications*, vol. 6, no. 03, p. 1450034, 2014.

[37] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2, pp. 199–219, 2006.

[38] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental sat solving," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[39] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *International Conference on Software Engineering*, 2015, pp. 540–550.

[40] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t)," in *International Conference on Automated Software Engineering*. IEEE, 2015, pp. 494–505.

[41] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Pacific Northwest Software Quality Conference*, 2006, pp. 419–430.

[42] D. R. Kuhn, R. Kacker, and Y. Lei, "Automated combinatorial test methods: Beyond pairwise testing," *Crosstalk, Journal of Defense Software Engineering*, vol. 21, no. 6, pp. 22–26, 2008.

[43] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[44] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 323–334.

[45] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on software engineering*, no. 4, pp. 438–444, 1984.

[46] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing (keynote paper)," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2015, pp. 1–12.

[47] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 255–264.

[48] X. Qu and M. B. Cohen, "A study in prioritization for higher strength combinatorial testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 285–294.

[49] M. Papadakis, C. Henard, and Y. Le Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 1–10.

[50] E.-H. Choi, S. Kawabata, O. Mizuno, C. Artho, and T. Kitamura, "Test effectiveness evaluation of prioritized combinatorial testing: a case study," in *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, 2016, pp. 61–68.

[51] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 47–60.

[52] T. Y. Chen and Y.-T. Yu, "On the relationship between partition and random testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, 1994.

[53] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *International Conference on Software Engineering*. IEEE, 2011, pp. 1–10.

[54] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2012, pp. 1–59.

[55] Wikipedia, "Tukey's range test," https://en.wikipedia.org/wiki/Tukey's_range_test.

[56] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[57] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. Porter, "Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 43–66, 2014.