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.2019.2955687, IEEE Transactions on Software Engineering

Comparative Analysis of Constraint Handling Techniques for Constrained Combinatorial Testing

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

Abstract—Constraints depict the dependency relationships between parameters in a software system under test. Because almost all systems are constrained in some way, techniques that adequately cater for constraints have become a crucial factor for adoption, deployment and exploitation of Combinatorial Testing (CT). Currently, despite a variety of different constraint handling techniques available, the relationship between these techniques and the generation algorithms that use them remains unknown, yielding an important gap and pressing concern in the literature of constrained combination testing. In this paper, we present a comparative empirical study to investigate the impact of four common constraint handling techniques on the performance of six representative (greedy and search-based) test suite generation algorithms. The results reveal that the *Verify* technique implemented with the Minimal Forbidden Tuple (MFT) approach is the fastest, while the *Replace* technique is promising for producing the smallest constrained covering arrays, especially for algorithms that construct test cases one-at-a-time. The results also show that there is an interplay between effectiveness of the constraint handler and the test suite generation algorithm into which it is developed.

Index Terms—combinatorial testing, constraint, survey, comparative study

1 Introduction

Combinatorial testing (CT), or combinatorial interaction testing (CIT), is a potentially powerful testing technique for revealing failures triggered by interactions of parameters that govern software system execution behaviour [1]. Since the initial idea of CT was sketched in 1985 [2], it has been an active research area with over 760 scientific publications[1], contributing to the development of theory, techniques and applications. CT is also gradually finding its way into industrial practice, and has been included into testing standards such as ISO/IEC/IEEE 29119 [3].

Traditionally, CT assumes that the parameters of software under test are independent from each other. The τ-way covering array, in which each input combination of τ parameters must appear at least once, can thus be directly used as the test suite. However, in real-world programs, there usually exist dependency relationships between parameters. Such relationships can be described as constraints in CT, indicating that some particular input combinations are infeasible or undesirable. Any application of a τ-way covering array that fails to take constraints into account will lead to many 'invalid' test cases. As a result, CT could be less effective than people would otherwise expect.

For example, a constraint may indicate that the Linux operating system cannot be combined with the IE browser: if a testing strategy requests this combination, then no test case can realise it. Of course, we could simply dismiss such invalid combinations, but there is a computational cost in doing so and the result may also affect the size of the test suites; perhaps other valid sets of combinations could be reduced if the constraint was handled earlier.

Constraints typically denote tests that simply cannot be achieved. While these may be an irritation, more pernicious are hidden constraints. For example, sometimes a test case that violates constraints may still be executed, but will yield results that are difficult to distinguish from a software failure. Where such constraints are merely implicit, they can lead to considerable wasted effort generating and analysing results from tests that should have been avoided from the outset.

The concept of a *constraint* was introduced early in the development of CT. In 1987, Tatsumi et al. were already aware that "not all of the combinations of the factors and states entered in the test factor table actually exist" and "these conditions deserve special consideration" [4], [5]. The early CT tools of 1990s, such as CATS [6] and AETG [7], were also able to deal with constraints. In 2006, Grindal et al. [8] presented the first review of constraint handling techniques. In their study, constraints are assumed to be few and simple in practice, so the techniques to modify test models to remove constraints played an important role at that time.

However, in 2007, the study by Cohen et al. revealed that constraints are large and complex in real-world applications, especially for highly configurable software systems [9]. They found that the previous constraint handling techniques were

- *H. Wu and C. Nie are with Department of Computer Science and Technology, Nanjing University, Nanjing, China, 210023. E-mail: {hywu, 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 April XX, XXXX; revised August XX, XXXX.

1. The data to support this claim was obtained from Combinatorial Testing Repository (http://gist.nju.edu.cn/ct_repository).

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

unsatisfactory in terms of both generality and scalability, and proposed to use a satisfiability (SAT) solver as a general solution to resolve constraints during test suite generation. Since then, a variety of automated constraint handling techniques have been developed and successfully applied [10], [11]. New research directions, such as automatically inferring [12], validating [13], [14], and repairing constraints [15], have also started to attract attention.

Even though the topic of constraints has been discussed in many studies since 1987, the importance of constraints, and the need for constrained combinatorial testing, remains a fundamental barrier to the wider update of CT. In 2014, Khalsa and Labiche [16] analysed the applicability of 75 test suite generation algorithms and tools in CT. They found that more than half of these studies simply do not implement any constraint handling technique. More recently, Wu et al. [11] further examined the test suite generation publications between 2015 and 2018. They found that the influence of constraints is only accounted in 30% of them, which makes a large proportion of the proposed techniques inapplicable to many real-world programs.

Moreover, since recent studies [10], [11] have revealed a diverse set of different constraint handling techniques, a subsequent natural question is the extent to which these techniques make a difference in the performance of different test suite generation algorithms. This is important because test suite generation studies of CT usually follow (arbitrary choices of) one of the previous practices to determine the constraint handler to be used (for example, many recent studies directly use the SAT solver approach without further examination [17], [18], [19], [20], [21]), which may potentially restrict the performance of the proposed algorithms. Unfortunately, except one early comparative study [8] that is based on unrealistic settings (few and simple constraints), none of the studies has presented results on the relationship between the more recently developed constraint handling techniques and test suite generation algorithms.

In this study, we present the first comparative analysis to investigate the impact of choices concerning constraint handling techniques and test suite generation algorithms that use them. To establish a uniform (level playing field) empirical comparison, we first developed a framework of constrained covering array generation as the reference implementation, which includes the four common constraint handling techniques (including *Verify*, *Solver*, *Tolerate*, and *Replace*) as configurable options. A total number of six representative greedy and search-based test suite generation algorithms (including AETG [22], DDA [23], IPO [24], PSO [25], SA [26], and TS [27]) were then implemented based on this framework, and the experiments were conducted on a well-known benchmark [22] of constrained covering array generation. Finally, the sizes of test suites, computational costs, and failure revelation abilities obtained from different combinations of test suite generation algorithms and constraint handlers are recorded and analysed.

Our experimental results reveal that the constraint handlers denote a key decision point when designing new test suite generation algorithms, instead of simply treating the solver as an afterthought or a detachable independent component. It is important to take both generation algorithms and test goals into consideration to determine the 'best' constraint handler that should be used.

This study seeks to address the important gap of adequately handling constraints in test suite generation for CT, and thereby extends current knowledge on constrained combinatorial testing. We hope and believe that this work may be helpful to facilitate future research and practice in constrained combinatorial testing, as well as to suggest potentially promising constraint handling techniques for the design of test suite generation algorithms.

Summing up, the primary findings of this study are as follows:

1) We observe statistically significant difference in performance in not only test suite size (86% of cases) and computational cost (99% of cases), but also failure revelation ability (52% of cases), when different constraint handling techniques are used in the test suite generation algorithm.
2) The technique that resolves constraints after test suite generation, i.e., *Replace*, performs surprisingly well, confounding the 'conventional wisdom' that prefers techniques that avoid constraints during test suite generation. Especially for generation algorithms that construct test cases one-at-a-time, *Replace* can produce significantly smaller test suites and the $\widehat{A}_{12}$ effect size is higher than 0.9 in 70% of cases, indicating a very high probability that *Replace* is highly effective. Whereas, when the whole test suite is directly constructed by search-based algorithms, *Tolerate* could be a more promising choice, as large effect sizes (higher than 0.8 or lower than 0.2) are observed in 71% of cases for which *Tolerate* is significantly better.
3) The *Verify* technique implemented with the Minimal Forbidden Tuple (MFT) approach tends to be the fastest technique, while the technique that relies on a constraint satisfaction solver, i.e., *Solver*, is usually the slowest. In particular, for the greedy algorithms (AETG, DDA and IPO), the average $\widehat{A}_{12}$ effect size between *Verify* and the other techniques is only 0.07; there is little, or no chance that *Verify* could perform worse than its competitors.
4) The constraint handling technique that minimises the size of τ-way covering arrays tends to result in a lower chance of detecting failures triggered by $k > \tau$ parameters, but the observed difference is not as large as that in the size of test suites. Especially, the large difference (with $\widehat{A}_{12}$ higher than 0.8 or lower than 0.2) in test suite size can be observed in 42% of cases, while this proportion is, on average, only 10% in terms of failure revelation ability.

The rest of this paper is organised as follows. Section 2 sets out the background on constrained combinatorial testing, and presents a brief review of currently available constraint handling techniques. Section 3 explains the experiment methodology of our comparative analysis of constraint handling techniques. Section 4 presents experimental findings and discussions. Section 5 analyses threats to validity, and finally, Section 6 concludes this work.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

TABLE 1
A test model for 'font effect'.

Style	Underline	Underline Colour	Superscript	Subscript
Regular	On	Red	On	On
Italic	Off	Blue	Off	Off
Bold		Green		

Constraint: *Superscript* and *Subscript* cannot both be enabled

2 Constrained Combinatorial Testing

We begin this study with background on constrained combinatorial testing, and also a brief review of currently available constraint handling techniques.

2.1 Background

Combinatorial testing (also known as combinatorial interaction testing) is a systematic technique that selects combinations of program inputs or features for testing [28]. It models the input space of the Software Under Test (SUT) by a set of n parameters (such as system configurations, internal or external events, and user inputs) and their associated value domains (a finite set of discrete values). A *test case* of the SUT is then produced by assigning to each parameter a specific value [1].

Table 1, for example, shows a test model for testing font effects in a word processor (this example is adapted from previous work [29]). This test model has five parameters: *Style* and *Underline Colour* can take three values, and each of the others can take two values.

Software failure in such systems is usually triggered by interactions of parameters, which can be represented using a τ-way combination (i.e., a combination of τ parameter values). Here, we refer to a failure as the inability of a system to perform its required functions; while a fault is a manifestation of an error, which is a human action that produces an incorrect result [30].

Exhaustive testing covers all n-way combinations by definition. Such test suites, however, are usually prohibitively large. Instead, CT provides a systematic approach to selecting a subset of all possible inputs by using a a mathematical object named a τ-way covering array. Its concept comes from the fact that if no more than τ parameters are involved in any failure, then covering all k-way combinations ($k \leq \tau$) is effectively equivalent to exhaustive testing.

Definition 1 (Covering Array [1]). A τ-way covering array of a SUT is a set of test cases, in which every τ-way combination is covered at least once. Such a covering array can be denoted by $CA(N; \tau, v_1^{k_1} v_2^{k_2} \ldots v_m^{k_m})$, where $v_i^{k_i}$ stands for k_i parameters with the same number of v_i values, $\sum k_i = n$.

The value of τ is referred to as the *strength* of a covering array. Determining this value is a key issue in CT. The empirical observations of Kuhn et al. have demonstrated that most software failures can be triggered by interactions of one or two parameters, and the value of τ is not likely to exceed six [31]. Hence, $\tau = 2$, or pairwise, is the most widely used choice in practice, which can achieve a good balance between test suite size and failure finding effectiveness.

TABLE 2
A 2-way constrained covering array $CA(9; 2, 3^1 2^1 3^1 2^2)$.

	Style	Underline	Underline Colour	Superscript	Subscript
t_1	Regular	On	Red	Off	On
t_2	Regular	Off	Blue	On	Off
t_3	Regular	On	Green	Off	Off
t_4	Italic	Off	Red	On	Off
t_5	Italic	On	Blue	Off	Off
t_6	Italic	Off	Green	Off	On
t_7	Bold	Off	Red	Off	On
t_8	Bold	Off	Blue	Off	On
t_9	Bold	On	Green	On	Off

The conventional definition of a covering array assumes that every possible τ-way combination is feasible and has the potential to trigger a failure. However, this may be unrealistic due to the constraints between parameter values. Constraints may be introduced because of inconsistencies between hardware components, limitations on possible configurations, or simply design choices [9]. For example, one constraint for the model in Table 1 is that "*Superscript* and *Subscript* cannot both be enabled for the same character". A test case that violates this constraint is considered invalid.

In order to incorporate constraints into CT, the definition of a covering array needs to be extended to that of a constrained covering array, which can be defined as follows:

Definition 2 (Constrained Covering Array [22]). A τ-way constrained covering array of a SUT with respect to a set of constraints C is a set of test cases, in which (1) each test case is C-satisfying; and (2) every C-satisfying τ-way combination is covered at least once.

Table 2 gives a 2-way constrained covering array of the test model in Table 1, where each row is a valid test case of the SUT (the invalid combination, *Superscript* $=$ *On* $\wedge$ *Subscript* $=$ *On*, does not appear in this table). Here, instead of exhaustively examining all 54 constraint satisfying test cases, CT only requires 9 test cases to cover every valid 2-way combination at least once.

Constraints in CT can be either *hard* or *soft*. A hard constraint requires that certain parameter combinations cannot appear in any test case, because their existence will prevent the test case from execution. The constraint in Table 1 is an example of a hard constraint. A soft constraint, on the other hand, is the combination that does not need to be tested, based on the knowledge and experience of testers [32]. It is possible to include test cases that violate soft constraints, but these are undesirable and bring no benefit to test effectiveness.

Intuitively, all constraints are explicitly specified in the test model. Sometimes the interactions of a set of constraints may give rise to new constraints. Such a newly introduced constraint is named an *implicit constraint*, because usually such implicit constraints are unknown to the tester. For example, for the test model in Table 1, we already have one invalid combination ($-$, $-$, $-$, *On*, *On*). If we add another

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

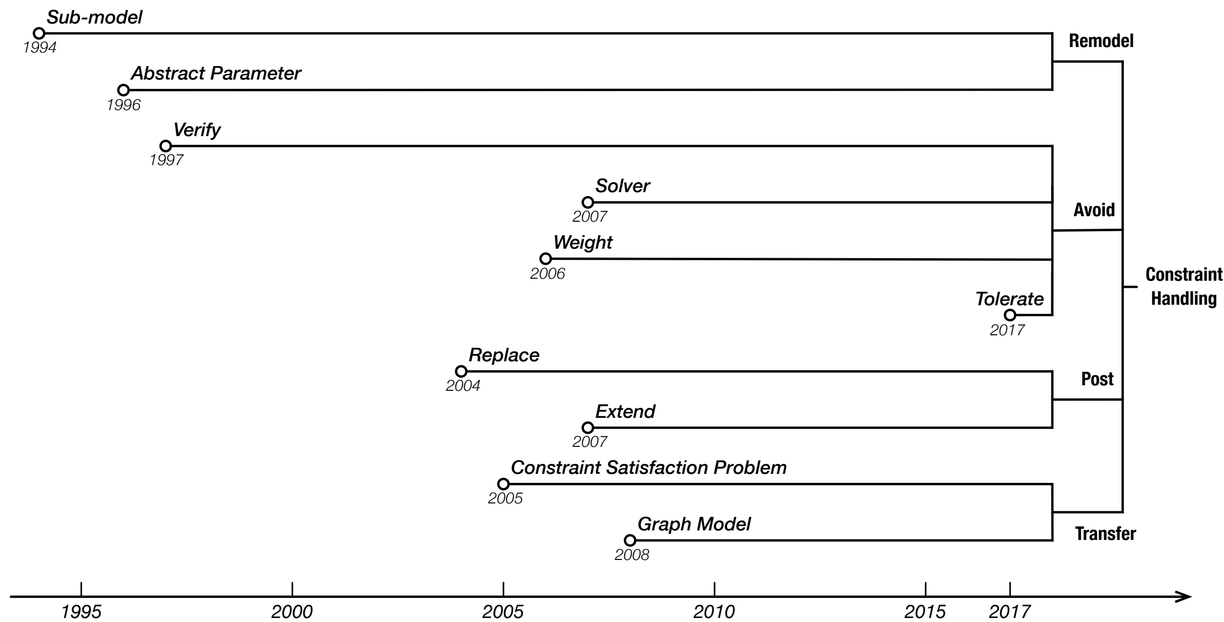

Fig. 1. The chronological development of constraint handling techniques [11].

invalid combination ($Bold$, $-$, $-$, *Off*, $-$) as a constraint, then a new invalid 2-way combination (*Bold*, $-$, $-$, $-$, *On*) is introduced, because we cannot find a test case that covers both this combination and does not cover the two explicitly given invalid combinations.

The impact of constraints may vary with different problems, but in general, constraints increase the complexity and difficulty of effectively applying combinatorial testing [11]. Nevertheless, testers can build more accurate and flexible test models for the SUT with the help of constraints. A large number of constraints can also greatly reduce the size of the search space, which makes the generation of high strength covering arrays feasible at a reasonable computational cost [33], [34].

2.2 Constraint Handling Techniques

Given a test model with constraints, one key challenge in CT is to generate a constrained covering array of the minimum size to cover all valid τ-way combinations. Constraint handling focuses on this process to ensure that the final solution only contains valid test cases. Currently, there are four main categories of constraint handling techniques that can be used [10], [11]: *Remodel*, *Avoid*, *Post-Process*, and *Transfer*. Figure 1 additionally gives an overview of the chronological development of these techniques [11].

2.2.1 Remodel

The 'remodel' technique focuses on eliminating constraints from test models before test suite generation, so that conventional covering arrays can be used directly. The ***Sub-model*** and ***Abstract Parameter*** techniques are two representative choices in this category, which remove constraints by constructing conflict-free sub-models [6], and combine conflicting parameters into abstract parameters [35], respectively. These techniques are typically less competitive in the size of generated test suite, and rely on manual efforts for the model modification [8], [36].

2.2.2 Avoid

The 'avoid' technique focuses on constructing conflict-free solutions during test suite generation. It typically integrates particular strategies as extensions into greedy or search-based generation algorithms (for example, to ensure that each parameter value assignment is constraint satisfying). There are four representative choices in this category:

1) ***Verify***. The 'verify' technique is probably the most basic technique to handle constraints in the more general case. Its idea is to maintain a list of forbidden tuples, so that each partial or complete solution during the generation process can be verified against them to prevent the appearance of constraint violation [7]. Recently, Yu et al. [37] improved this technique by introducing the concept of *Minimum Forbidden Tuple* (MFT). A MFT is a forbidden tuple of minimum size that covers no other forbidden tuples. Once all MFTs are found, validity verification can be quickly performed by only checking whether a solution contains any MFT.
2) ***Solver***. The idea of 'solver' technique is similar to *Verify*, but it encodes constraints and solutions into a formula and applies an existing constraint satisfaction solver (usually, a SAT solver) to check the formula's validity [22]. This technique can be integrated into any generation algorithm, but it may lead to inefficient implementations due to the large number of solver calls required. To address this issue, several improvements have been proposed, such as using the solvers only when necessary [9], [38], as well as exploiting the solving result and history [19], [22], [38].
3) ***Weight***. The 'weight' technique was initially developed to cater for soft constraints, where combinations were weighted as either important with positive values, or

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

undesirable with negative values [32]. This technique only avoids the undesirable combinations where possible, and does not guarantee the exclusion of such constraints [32].

4) ***Tolerate***. When using search-based algorithms to generate constrained covering arrays, one choice is to exclude all invalid solutions from the search space, so that all intermediate and final solutions are constraint satisfying. However, sometimes some elements of invalid intermediate solutions may help to find the optimal solution, so it maybe desirable to 'tolerate' such invalid solutions in the search space, but penalise them in favour of valid solutions [25], [27]. For example, one choice to implement this technique is to incorporate a penalty term into the fitness function, which evaluates the number of constraint violations (such a value can be calculated using the the minimal forbidden tuple approach or a constraint solver) [27].

2.2.3 Post-Process

The 'post-process' technique focuses on repairing constraint violations after test suite generation. Its aim is to resolve conflicts in covering arrays that are generated without considering constraints, while at the same time retaining combination coverage. This process usually starts from the identification of all invalid test cases, which can be efficiently achieved by using the minimal forbidden tuple approach or a constraint solver.

Currently, there are two representative techniques in this category. The ***Replace*** technique tries to replace invalid test cases by a set of valid ones [8], [39], which is general enough to be combined with any test suite generation algorithm. While the ***Extend*** technique is specifically designed for event sequence testing, which inserts new events to create executable test sequences to remove conflicts [40].

2.2.4 Transfer

The 'transfer' technique focuses on reformulating the problem of covering array generation into other problems, or using other structures to model the constrained input space, so that final solutions can be directly obtained by applying existing algorithms or tools. One choice of this technique is to reduce the covering array generation problem into the ***Constraint Satisfaction Problem (CSP)***, and then constraint solvers can be used to produce satisfiable test cases or test suites [41], [42]. Such CSP-based techniques can generate the covering array of minimum size and prove its optimality, but they usually have high computational cost, and are only practicable for pairwise testing [42], [43], [44].

Another choice is to use a ***Graph Model*** to represent and manipulate the search space. This allows graph-related operations and theories to be used directly to construct test cases or test suites. Exemplary applications include navigation graph [45], binary decision diagram [46], edge clique covering problem [47], and graph colouring problem [48].

3 Experimental Design

In this section, we describe the research questions we seek to investigate, and explain our experimental methodology.

3.1 Research Question

Constraint handling is the most prominent research field in constrained combinatorial testing [11]. With the rich collection of currently available constraint handling techniques (as reviewed in Section 2.2), there is clearly a need to understand how these techniques will make a difference in different test suite generation algorithms. This thereby motivates our main research question:

> *Given a particular test suite generation algorithm, is there a significant difference in performance between constraint handling techniques?*

Especially, we are interested in the *test suite size* ($\mathbf{RQ_1}$) and *computational cost* ($\mathbf{RQ_2}$) that can be achieved when using different constraint handlers. These are the conventional performance indicators for evaluating a test suite generation algorithm in CT. In addition, test practitioners might also be interested in the *failure revelation ability* ($\mathbf{RQ_3}$) of the τ-way covering arrays generated, in particular when a failure is triggered by combinations of more than τ parameters. We will investigate the performance of different constraint handlers in terms of these three aspects.

3.2 The CCAG Framework

In order to answer our research questions, we need to execute and compare a particular test suite generation algorithm with different choices of constraint handlers. Although there are available tools for test suite generation, for example, PICT[2], ACTS[3] and CASA[4], the selection of different constraint handlers is not supported in these tools. Moreover, these tools are implemented by different developers with different programming languages, which might otherwise introduce additional sources of bias into our evaluation. Therefore, in this study, we chose to develop a reference implementation, which we imagine to have been realised in any greedy or search-based constrained covering array generation algorithm (there is already an exemplary study in the fuzzing literature [49]).

Algorithm 1 The CCAG Framework

1: `PreProcess()`
2: $S \leftarrow$ an initial solution
3: **while** S is not a covering array **do**
4: $\quad S \leftarrow$ `Next` (S, `isValid()`, `PenaltyTerm()`)
5: **end while**
6: `PostProcess()`
7: **return** S

Algorithm 1 shows our Constrained Covering Array Generation (CCAG) framework. It is general enough to accommodate the three widely used frameworks for covering array generation [50]: (1) iteratively constructing a single test case that maximises combination coverage (*one-test-at-a-time*); (2) firstly constructing a test set for τ parameters and then extending it horizontally and vertically (*in-parameter-order*); and (3) directly constructing a covering array for

2. https://github.com/Microsoft/pict
3. https://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software
4. http://cse.unl.edu/~citportal/

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

TABLE 3
The detailed implementations of the constraint handler-related functions in the CCAG framework.

	`PreProcess`	`isValid`	`PenaltyTerm`	`PostProcess`
Verify	Calculate MFTs	Determine based on MFTs	Return zero	None
Solver	Initialise a SAT solver	Determine based on the solver	Return zero	None
Tolerate	Calculate MFTs	Return true	Calculate based on MFTs	None
Replace	Calculate MFTs	Return true	Return zero	Replace invalid test cases by valid ones

a given size (*evolve-test-suite*). These three frameworks can be realised by both greedy and search-based strategies. In either case, the generation algorithm will start from an initial solution S, and then apply a step-by-step process (i.e., the `Next` function) to improve S (typically, by assigning or changing parameter values), in order to cover as many combinations as possible. Finally, a covering array is achieved when all valid τ-way combinations are covered.

We then chose constraint handling techniques that can be incorporated into the CCAG framework. This excludes techniques in the *Transfer* category, because they use constraint solvers and graph-related techniques to construct solutions directly, which thus cannot be combined with algorithms that construct covering arrays step-by-step. Moreover, we want the techniques we consider to be automated and general (not designed for a specific algorithm or test scenario) to enable a fair comparison. This excludes techniques in the *Remodel* category, because they usually require manual effort to modify test models. This requirement also excludes *Weight* and *Extend* techniques, because they can only be used to deal with soft constraints, and test sequences in GUI testing, respectively, while our focus is to handle the more common hard constraints in general cases. Consequently, *Verify*, *Solver*, *Tolerate*, and *Replace* are the only feasible choices among all reviewed techniques in Section 2.2.

To incorporate the above four constraint handling techniques, CCAG provides a uniform interface that consists of the following four functions: `PreProcess`, `PostProcess`, `isValid`, and `PenaltyTerm`. As long as a generation algorithm is implemented under the CCAG framework with these functions, it can be easily configured to work with any of the four constraint handlers. Specifically,

- `PreProcess` is the process executed before executing the covering array generation algorithm, such as configuring and initialising the constraint handler based on the given test model.
- `isValid` takes a τ-way combination, a test case, or a test suite as input, and returns a Boolean value indicating whether the given candidate solution is constraint satisfying.
- `PenaltyTerm` takes a test case, or a test suite as input, and returns the value of the penalty term for calculating the fitness function of the candidate solution (in particular, for the *Tolerate* technique).
- `PostProcess` is the process executed once a covering array is generated.

Different constraint handling techniques typically perform different tasks in these functions. We will explain the respective implementations of the four constraint handling techniques in the next section.

3.3 Constraint Handling Techniques

Table 3 shows the detailed implementations of the four constraint handling techniques that the CCAG framework includes.

The *Verify* technique uses the Minimal Forbidden Tuple (MFT) approach to resolve constraints by definition. It thus needs to calculate the set of MFTs in `PreProcess`. We implemented the same approach, as reported in previous work [37], to deduce the set of MFTs. The validity of a candidate solution can then be determined by verifying against MFTs in `isValid`. In addition, as *Verify* does not allow invalid intermediate solutions, the `PenaltyTerm` will always return zero (namely, there is no penalty term in the fitness function). There is also no task to perform in `PostProcess`.

The implementation of the *Solver* technique is similar to that of *Verify*, but it uses a constraint solver to resolve constraints, instead of using the MFT approach. Here, we used SAT4J[5], which is a widely used constraint solver for JAVA, to implement *Solver*. Accordingly, it needs to initialise the solver in `PreProcess`, and the validity of a candidate solution is determined by solving a constraint satisfaction problem in `isValid`.

For *Tolerate* and *Replace*, one key operation is to determine whether a constraint is violated in a candidate solution. This can be implemented by using either the MFT approach or a constraint satisfaction solver. Here, we choose to use the same MFT approach [37], as used in the *Verify* technique. As a result, these two techniques need to calculate the set of MFTs in `PreProcess`. Moreover, because *Tolerate* allows invalid intermediate solutions, and *Replace* simply ignores all constraints during the generation, `isValid` will always return *True* in these two techniques.

The *Tolerate* technique incorporates a penalty term into the fitness function to include invalid intermediate solutions into the search space. The fitness function used in this study is of the same form as proposed in previous work [27]:

$$fitness(s) = U(s) + \omega \times V(s)$$

where s is a candidate solution, $U(s)$ measures the ability of s in covering τ-way combinations, $V(s)$ is the number of constraint violations in s, and ω is the penalty weight. Here, if s indicates a test suite (i.e., the *evolve-test-suite* framework), we define $U(s)$ as the number of uncovered combinations in s (the goal is to minimise the fitness function); otherwise, $U(s)$ is the number of uncovered combinations that can be covered by s (the goal is to maximise the fitness function). In either case, the `PenaltyTerm` of *Tolerate* will calculate and return the value of $V(s)$.

5. http://www.sat4j.org

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

Algorithm 2 The Replace Algorithm

1: T = a τ-way covering array generated without taking constraints into account
2: $R = \emptyset$
3: **for** each invalid test case $t \in T$ **do**
4: X = the set of valid τ-way combinations that are only covered by t
5: $R = R \cup X$
6: remove t from T
7: **end for**
8: $E = \emptyset$
9: **for** each τ-way combination $x \in R$ **do**
10: C = the set of incomplete test cases in E that are both compatible and constraint satisfying with x
11: **if** $C \neq \emptyset$ **then**
12: use x to update $c \in C$, such that c has the largest number of overlapping fixed parameters with x
13: **else**
14: add x into E
15: **end if**
16: **end for**
17: **for** each incomplete test case $e \in E$ **do**
18: assign unfixed parameters of e values chosen at random from those that are constraint satisfying
19: **end for**
20: $T = T \cup E$
21: **return** T

Unlike the above three constraint handling techniques, *Replace* resolves constraints in `PostProcess`. A straightforward approach is to replace each invalid test case by a specific set of valid test cases, selecting to retain combination coverage. Because this approach tends to lead to unnecessarily large test suites, we used an alternative approach to replace test cases, as shown in Algorithm 2. The key idea of our approach is to combine as many 'compatible' tuples as possible into each test case. Two tuples are compatible if values from both tuples are the same or at least one of them is unfixed (note that a τ-way combination is a n-tuple with τ fixed parameters). For example, tuples $(1, 1, 1, -)$ and $(1, -, -, 2)$ are compatible, but tuples $(1, 1, 1, -)$ and $(1, 2, -, -)$ are not.

In Algorithm 2, we firstly determine R: the set of valid τ-way combinations that are only covered by invalid test cases, and remove all invalid test cases from T (Lines 2-7). We then generate a set of test cases, E, to cover all combinations in R (Lines 8-16).

For each valid τ-way combination x in R, we find all incomplete test cases (n-tuples with unfixed parameters) that can cover x from E, namely each of these test cases should be compatible with x, and also constraint satisfying if it is updated with x. If there are such test cases, we select the one that has the largest number of overlapping fixed parameters with x, and update it with x (we assign values to as few unfixed parameters as possible); otherwise, as x cannot be covered by any of the existing test cases in E, we add x to E.

After that, if there remain unfixed parameters in E, these parameters will be assigned to values chosen at random from the set of those that are constraint satisfying, so that each row of E is a complete and valid test case (Lines 17-19). Note that each incomplete test case in E is always constraint satisfying in this algorithm, so that we can find at least one valid value assignment. Finally, we combine test cases in T and E as the final constrained τ-way covering array.

3.4 Test Suite Generation Algorithms

We implemented six well-known covering array generation algorithms in this study: AETG [22], DDA [23], IPO [24], PSO [25], SA [26] and TS [27].

We chose these algorithms because they are the most representative implementations of the three available covering array generation frameworks: *one-test-at-a-time*, *in-parameter-order* and *evolve-test-suite* [50]. They also cover the widely used computational search methods (greedy and heuristic search-based) to generate covering arrays. We do not consider mathematical methods, because they can only be used in a restricted subset of cases, though they can yield the optimal covering arrays [1].

Specifically, AETG [22], DDA [23] and PSO [25] are based on the *one-test-at-a-time* framework, during which a test case that covers the largest number of yet uncovered τ-way combinations is generated and added into the test suite one-at-a-time. AETG [22] and DDA [23] apply greedy strategies to construct such a test case, while PSO [25] applies the search-based particle swarm optimisation to achieve the same purpose.

IPO [24] is an implementation of the *in-parameter-order* framework. It first constructs a test set for τ parameters that have the largest number of values, and then conducts horizontal and vertical extensions for each of the remaining unfixed parameters. A greedy strategy is used to determine the best value assignment for each parameter.

SA [26] and TS [27] are based on the *evolve-test-suite* framework. They directly construct a covering array of size N, while the value of N is determined by a binary-search-like method [26]. For each choice of N, a random array A of size $N \times n$ is first initialised. This array is then evolved by the simulated annealing and tabu search strategies, guided by the fitness function that calculates the number of uncovered τ-way combinations in A.

Note that the performance of these test suite generation algorithms is usually impacted by their parameter settings. Here, we used the same settings as reported in their respective previous work [22], [23], [24], [25], [26], [27]. However, when the *Tolerate* constraint handler is used, our preliminary experiments reveal that SA and PSO usually need different settings for the number of iterations and penalty weight (ω), respectively. In order to ensure the quality of solutions, we set the maximum number of iterations in SA to 200000, and set the penalty weight in PSO to -0.05.

In addition, as our goal is to compare different constraint handling techniques, not to design the most effective generation algorithms, we only implemented the basic versions of the above six algorithms. As a result, the covering arrays generated in this study may be slightly larger than those reported in previous studies [22], [23], [24], [25], [26], [27].

3.5 Subjects and Process

We used a well-known benchmark of constrained covering array generation [22] as our subject test models. This bench-

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

mark consists of 35 test models; five of them are real-world problem instances, and the others are synthetic instances that are randomly generated to mimic the structures of the five real-world models. This set of models is often relied upon as a “standard benchmark set” to evaluate the performance of test suite generation [17], [18], [22], [26], [27].

For each test model, we used a total of 21 algorithms to generate 2-way constrained covering arrays, which are the most widely used objects in practice. Each of these algorithms is a combination of a test suite generation algorithm and a constraint handling technique: three variants (X+Verify, X+Solver, X+Replace) for each of AETG, DDA and IPO, and four variants (X+Verify, X+Solver, X+Tolerate, X+Replace) for each of PSO, SA and TS (note that *Tolerate* can be used only with a search-based algorithm). For each algorithm, the size of the test suite generated and its computational cost are directly recorded.

Regarding the failure revealing ability, because there is no available fault corpus of the subject test models used in this study, we chose to conduct simulation experiments to compare constraint handlers in terms of their failure finding effectiveness. Specifically, it is known that using a 2-way covering array can always detect a failure that is triggered by one or two parameters (in this case, a smaller test suite is more computationally cost-effective). While the remaining question is how different algorithms perform when a failure is triggered by the combinations of more than two parameters. To this end, for each test model, we first generate 100 failure-causing combinations for each of strengths 3, 4, 5, and 6 at random. The proportion of these combinations that can be “hit” by each algorithm is then recorded.

All variants of generation algorithms implemented in this study involve some level of random selection among candidate partial solutions[6]. Inferential statistical analysis is thus required to cater for the randomness in these algorithms [51], [52]. For each test model, each variant of generation algorithms is repeated 30 times. We then performed ANOVA (with significance level α set at 0.05) to determine whether there is a significant difference in performance of any pair of constraint handling techniques. We also performed the non-parametric Mann-Whitney U-test ($\alpha = 0.05$) for each pair of constraint handling techniques to further investigate their relationships.

Moreover, since performing only significance tests would be insufficient to measure the effect size (and thereby acceptability) of the findings, we also calculated the non-parametric Vargha and Delaneys $\widehat{A}_{12}$ effect size assessment for each pair of constraint handling techniques to investigate the magnitude of difference. Here, $\widehat{A}_{12}$ indicates the probability that one algorithm outperforms another. $\widehat{A}_{12} = 0.5$ denotes that algorithms 1 and 2 are equally likely to outperform one another, and the greater the $\widehat{A}_{12}$ the higher probability that running Algorithm 1 yields higher performance measure values (test suite size, computational cost, proportion of failures detected) than running Algorithm 2.

6. The original version of IPO [24] is a deterministic algorithm, where it assigns specific values to positions that have no impact on the combination coverage. In this study, we assign random values to such positions to introduce some level of randomness into this algorithm.

TABLE 4
Sizes of 2-way constrained covering arrays generated by different combinations of covering array generation algorithms and constraint handling techniques.

	Verify	Solver	Tolerate	Replace	# Diff
AETG	1588.6	1590.6	–	1399.9	34
DDA	1384.1	1382.7	–	1350.5	24
IPO	1276.0	1276.0	–	1290.6	29
PSO	1599.1	1597.2	1582.8	1399.9	34
SA	1451.5	1449.2	1286.6	1277.2	28
TS	1241.9	1247.1	1152.7	1204.2	31

TABLE 5
Computational cost (seconds) of the covering array generation algorithms with different constraint handling techniques for $\tau = 2$.

	Verify	Solver	Tolerate	Replace	# Diff
AETG	232.7	2522.8	–	350.7	35
DDA	85.0	482.8	–	229.4	35
IPO	12.2	21.1	–	163.9	35
PSO	1089.0	14732.2	580.2	652.9	35
SA	6126.8	10675.7	8124.8	5199.5	35
TS	20258.3	26146.8	19753.7	18395.5	33

Note that the aim of this study is to compare different constraint handling techniques for each test suite generation algorithm. We do not seek to compare the generation algorithms, as search-based algorithms usually produce smaller covering arrays than greedy algorithms [17], [18], [26], [27].

We implemented the CCAG framework as an open source project in JAVA (JDK 1.8), which allows other researchers to build on our reference implementation in subsequent comparisons, and to seek to replicate our results and findings. We also implemented the six test suite generation algorithms in CCAG for illustration and comparison. The experiment is executed on a machine with Intel Xeon CPU E5-2640 2.0GHz and 16GB RAM. The CCAG framework, all algorithms, and detailed experimental data can be obtained on this paper’s companion website: https://github.com/GIST-NJU/CCAG.

4 Results

This section presents the results of our experiments and answers the research questions.

4.1 The Difference in Performance

Tables 4, 5 and 6 show the sizes of test suites, computational costs, and average proportions of k-way failures detected obtained from all 35 test models, for each combination of test suite generation algorithms and constraint handling techniques. Especially, the last column of these tables reports the number of test models in which ANOVA produces a p-value < 0.05, namely, we have no evidence to claim that the different constraint handlers are drawn from the same performance distribution (the Null Hypothesis).

From Tables 4 and 5, among all 210 cases studied (6 generation algorithms $\times$ 35 test models), there are 180 (86%) cases where we have evidence that different constraint handling techniques exhibit statistically significant difference in

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

TABLE 6
Average proportion of k-way failures detected by different combinations of covering array generation algorithms and constraint handling techniques.

		Verify	Solver	Tolerate	Replace	# Diff
$k = 3$	AETG	0.97	0.97	–	0.96	21
	DDA	0.96	0.96	–	0.96	9
	IPO	0.95	0.95	–	0.96	19
	PSO	0.97	0.97	0.97	0.96	17
	SA	0.95	0.95	0.95	0.95	11
	TS	0.95	0.95	0.95	0.95	14
$k = 4$	AETG	0.85	0.85	–	0.82	24
	DDA	0.83	0.83	–	0.82	9
	IPO	0.80	0.80	–	0.81	22
	PSO	0.85	0.85	0.85	0.82	28
	SA	0.81	0.81	0.80	0.80	13
	TS	0.79	0.79	0.79	0.79	17
$k = 5$	AETG	0.64	0.64	–	0.60	25
	DDA	0.60	0.60	–	0.59	12
	IPO	0.56	0.56	–	0.57	28
	PSO	0.64	0.64	0.64	0.59	29
	SA	0.59	0.59	0.57	0.57	16
	TS	0.56	0.56	0.55	0.55	19
$k = 6$	AETG	0.42	0.41	–	0.38	25
	DDA	0.38	0.38	–	0.37	4
	IPO	0.34	0.34	–	0.35	23
	PSO	0.42	0.42	0.41	0.38	27
	SA	0.37	0.37	0.36	0.35	13
	TS	0.35	0.35	0.33	0.34	15

performance in terms of test suite size (i.e., p-value < 0.05). In almost all of these cases (99%), we also have evidence to support the claim that the different techniques will lead to significantly different computational costs.

In addition, from Table 6, there is no surprise that the proportion of failures detected decreases with the increase in the number of parameters involved in the failure-causing combinations (as the difficulty to cover such combinations increases). Nevertheless, when a failure is triggered by 3, 4, 5, and 6 parameters (i.e., the value of k), there are 43%, 54%, 61%, and 51% of cases where we have evidence that different constraint handling techniques exhibit significantly different failure revelation abilities, respectively.

These finding indicates that the constraint handler is indeed a crucial factor that impacts the performance of the test suite generation algorithms that use it. Both effectiveness and efficiency of an algorithm can be greatly improved with an appropriate choice of the constraint handler. For example, when AETG is used to generate 2-way covering arrays for these 35 test models, we can see that using *Replace* as the constraint handler will produce smaller test suites (12% reduction) with much lower time cost (7.2 times faster) than using *Solver* as in its original version [22].

Next, we determine which constraint handling technique performs best for each of the six test suite generation algorithms. For variants of each algorithm, we conducted Mann-Whitney U-test for each pair of constraint handlers. Tables 7, 8 and 9 show the number of cases where constraint handler A is significantly superior ($+$), significantly indistinguishable ($=$), or significantly inferior ($-$) to constraint handler B across all 35 test models, in terms of test suite size, computational cost, and failure revelation ability, respectively.

Correspondingly, Figures 2, 3 and 4 show the distributions of $\widehat{A}_{12}$ statistics for each pair of constraint handlers (namely, each box contains 35 effect sizes obtained from 35 test models). As we seek to generate smaller test suites faster, a higher $\widehat{A}_{12}$ indicates that the second constraint handler has a higher chance of outperforming the first one in terms of test suite size, or computational cost; for failure revelation, a higher $\widehat{A}_{12}$ indicates that the first constraint handler has a higher chance of finding more failures.

4.2 Size of Test Suite (RQ_1)

From Table 4, Table 7 and Figure 2, we can see that *Verify* and *Solver* generate test suites of comparable sizes in almost all cases. In particular, among all six generation algorithms, their performance is significantly indistinguishable in at least 94% (33/35) of cases, and the medians of effect sizes are very close to 0.5. This is because these two techniques are only responsible for the validity-checking of each value assignment, while the value to be assigned is determined by the greedy or search-based strategies embedded in the generation algorithm.

By comparison with *Verify* and *Solver*, *Replace* removes conflicts after test suite generation. From Table 4, Table 7 and Figure 2, we can see that this technique performs well in terms of test suite size. Especially for AETG, DDA, and PSO (the one-test-at-at-time based algorithms), it produces significantly smaller covering arrays than *Verify* and *Solver* in 33, 21, and 34 out of 35 cases, respectively; among which its chance of outperforming the other two techniques is higher than 0.9 in 70% of cases. Regarding the other algorithms, although the performance of *Replace* is not as good as that of the one-test-at-a-time variants, it can still significantly outperform *Verify* and *Solver* in at least five cases, in which the effect sizes are all higher than 0.6.

This finding confounds the 'conventional wisdom' that handling constraints as a post-processing phase is an ineffective approach, since there is no strong evidence that supports either computational cost-effectiveness or failure revelation of such a technique [11]. In particular, there is only one study in 2006 that examines the performance of *Replace* [8]. The results suggest that this technique can be a general choice, but it cannot produce smaller test suites than the techniques that avoid constraints during test suite generation. As a result, post-processing related techniques are less studied in the literature, and also not the primary choice for the design of new generation algorithms. For example, a number of recently developed algorithms [17], [18], [27] can produce very small test suites, while none of them uses the *Replace*, or similar, technique to resolve constraints.

Tolerate is a constraint handling technique for search-based algorithms. From Table 4, Table 7 and Figure 2, we can see that it is not significantly worse than either *Verify* or *Solver* in at least 86% (30/35) of cases, and the medians of effect sizes are close to 0.5 between *Tolerate* and these two techniques. However, the performance of *Tolerate* varies when comparing with *Replace*. For the one-test-at-a-time based PSO, it is significantly worse than *Replace* in 34 cases; while for the evolve-test-suite based SA and TS,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

TABLE 7
The number of test models where constraint handler A is significantly superior (+), indistinguishable (=), or significantly inferior (−) to constraint handler B in terms of test suite size.

A & B	Verify & Solver			Verify & Tolerate			Verify & Replace			Solver & Tolerate			Solver & Replace			Tolerate & Replace		
	+	=	−	+	=	−	+	=	−	+	=	−	+	=	−	+	=	−
AETG	0	34	1	–	–	–	1	1	33	–	–	–	1	1	33	–	–	–
DDA	0	35	0	–	–	–	3	10	22	–	–	–	3	11	21	–	–	–
IPO	0	35	0	–	–	–	19	6	10	–	–	–	19	6	10	–	–	–
PSO	0	34	1	4	23	8	0	1	34	3	23	9	0	1	34	0	1	34
SA	0	33	2	2	22	11	14	9	12	5	19	11	14	8	13	18	8	9
TS	2	33	0	0	28	7	25	5	5	0	26	9	25	5	5	25	7	3

TABLE 8
The number of test models where constraint handler A is significantly superior (+), indistinguishable (=), or significantly inferior (−) to constraint handler B in terms of computational cost.

A & B	Verify & Solver			Verify & Tolerate			Verify & Replace			Solver & Tolerate			Solver & Replace			Tolerate & Replace		
	+	=	−	+	=	−	+	=	−	+	=	−	+	=	−	+	=	−
AETG	35	0	0	–	–	–	23	4	8	–	–	–	0	0	35	–	–	–
DDA	34	0	1	–	–	–	31	3	1	–	–	–	2	0	33	–	–	–
IPO	34	0	1	–	–	–	35	0	0	–	–	–	30	0	5	–	–	–
PSO	35	0	0	0	0	35	0	0	35	0	0	35	0	0	35	15	9	11
SA	35	0	0	27	8	0	5	16	14	6	3	26	0	2	33	2	3	30
TS	32	3	0	12	18	5	6	14	15	0	2	33	1	4	30	4	15	16

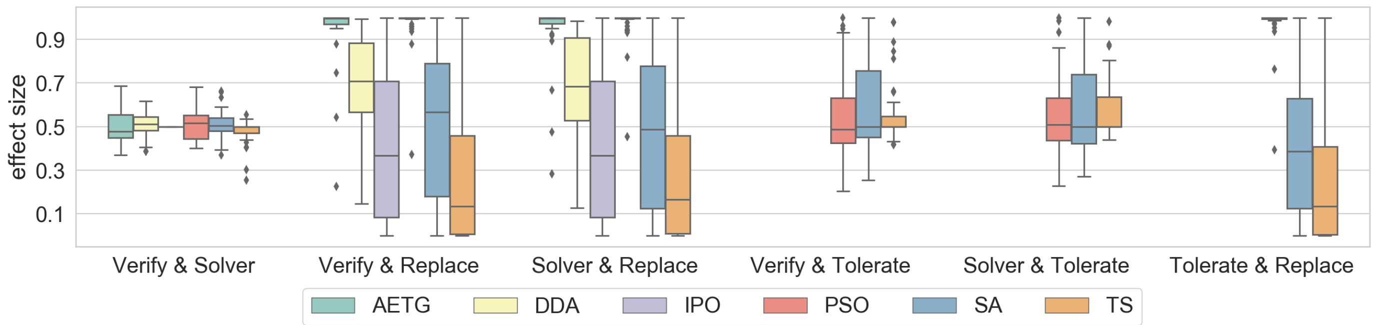

Fig. 2. The distribution of $\widehat{A}_{12}$ statistics for each pair of constraint handling techniques in terms of test suite size.

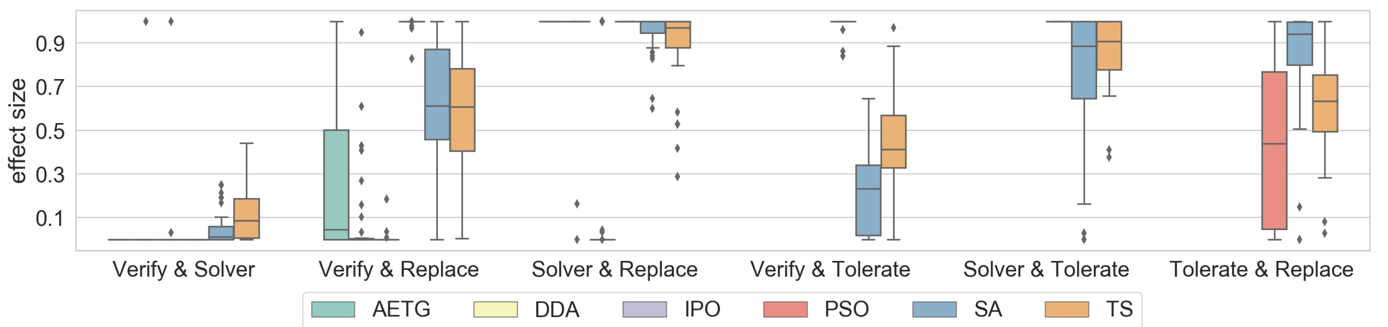

Fig. 3. The distribution of $\widehat{A}_{12}$ statistics for each pair of constraint handling techniques in terms of computational cost.

it outperforms in 18 and 25 cases, respectively. Moreover, among the cases where *Tolerate* performs significantly better for SA and TS, we observe large differences (with $\widehat{A}_{12}$ higher than 0.8 or lower than 0.2) in 71% of cases. This indicates that *Tolerate* has the potential to be an effective constraint handling technique for the search-based algorithms that directly construct the whole test suite.

Summing up, the experimental results in this section ($\mathbf{RQ_1}$) reveal that the *Replace* technique is the best choice to produce small test suites for generation algorithms of the one-test-at-a-time framework. The $\widehat{A}_{12}$ effect size can be higher than 0.9 in 70% of cases for AETG, DDA, and PSO. Whereas, for search-based generation algorithms of the evolve-test-suite framework, *Tolerate* could be a more

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

promising choice, as large effect sizes (higher than 0.8 or lower than 0.2) can be observed in 71% of cases for which *Tolerate* is significantly better.

4.3 Computational Cost ($\mathbf{RQ_2}$)

As far as the computational cost is concerned, from Table 5, Table 8 and Figure 3, we can see that *Solver* tends to be the most time consuming technique. Specifically, for all generation algorithms except IPO, the probability that *Solver* is slower than *Verify*, *Replace* and *Tolerate* is at least 0.8 in 93%, 94% and 77% of cases, respectively. While for IPO, although *Replace* usually spends the most time, the difference between any two techniques is nevertheless less than two seconds in 21 out of 35 cases.

By contrast, *Verify* tends to be the fastest constraint handling technique. Especially for AETG, DDA and IPO, the average $\widehat{A}_{12}$ effect size between *Verify* and the other techniques is 0.07, indicating a very high probability that *Verify* runs faster. However, for PSO, SA and TS, the computational cost of *Verify* can be significantly higher than that of *Replace* in 35, 14 and 15 cases, respectively. Moreover, for *Tolerate*, although it can produce smaller covering arrays than *Replace* for SA and TS, it is slower than *Replace* with a probability of higher than 0.63 in at least half of the cases. This indicates that *Replace* could be a time efficient choice to handle constraints for the search-based algorithms.

Note that the implementations of *Verify*, *Replace* and *Tolerate* are all based on the Minimal Forbidden Tuple (MFT) approach, namely, they use the set of MFTs to determine the validity of a candidate solution (see Table 3). Our finding thus indicates that using MFT for the validity-checking is much more efficient than using a constraint solver: for the MFT-based techniques, we only need to calculate MFT once, and check whether a forbidden tuple is involved at each value assignment; while for *Solver*, we need to solve a more complex constraint satisfaction problem on each occasion.

Summing up, the experimental results in this section ($\mathbf{RQ_2}$) reveal that the constraint handling technique that relies on a constraint satisfaction solver (i.e., *Solver*) is usually the slowest technique. By contrast, the *Verify* technique implemented with the Minimal Forbidden Tuple (MFT) approach tends to be the fastest technique for greedy algorithms (AETG, DDA, and IPO), as it has only a probability of 7% that spends more time than the other techniques. While for the search-based algorithms of the evolve-test-suite framework, the *Replace* technique could be a more time efficient choice.

4.4 Failure Revelation Ability ($\mathbf{RQ_3}$)

The last question concerns the failure revelation ability that can be achieved by different combinations of test suite generation algorithms and constraint handling techniques. From Table 6, Table 9, and Figure 4, we can see that test suites of comparable sizes usually exhibit similar performance in terms of failure revelation. Specifically, *Verify* and *Solver* are the two techniques that produce test suites of comparable sizes, and their proportions of failures detected are also significantly indistinguishable in at least 29 out of 35 cases (for any value of k).

The *Replace* technique is identified as the best choice to produce small 2-way covering arrays, especially for AETG, DDA, and PSO (the one-test-at-a-time based algorithms). But for failures that are triggered by $k > 2$ parameters, we find that there are only up to two cases where *Replace* can lead to significantly higher failure revelation ability for these three generation algorithms. Moreover, the number of cases where *Replace* performs significantly worse increases with the increase of k, achieving the maximal value when $k = 5$. For example, for AETG, the proportion of failures detected by *Replace* is significantly lower than that of *Verify* in 19, 24, 26 and 23 cases for $k = 3, 4, 5$ and 6, respectively. Similar findings can also be observed for other algorithms and the *Tolerate* constraint handling technique.

However, despite the fact that constraint handling techniques that produce smaller test suites tend to detect smaller number of failures, the magnitude of difference in failure revelation is not as big as that in the test suite size. Overall, for all combinations of generation algorithms and constraint handlers, we can observe large difference ($\widehat{A}_{12}$ effect size higher than 0.8 or lower than 0.2) in test suite size in 41% of cases. Whereas, the average chance that the larger test suite detects more failures is only 0.68 among those cases; and the large differences in detecting failures of strengths 3, 4, 5 and 6 are only observed in 5%, 11%, 14% and 11% of cases, respectively.

Summing up, the experimental results in this section ($\mathbf{RQ_3}$) reveal that the constraint handling technique that minimises the size of τ-way covering arrays will also lower its ability of detecting failures triggered by $k > \tau$ parameters, and this impact will increase with the increase of k. But for the cases where one constraint handler has a probability of higher than 0.8 to produce a smaller test suite, its average chance of detecting a lower proportion of failures is relatively low, only 0.68.

5 THREATS TO VALIDITY

As far as internal threats to validity are concerned, the performance of test suite generation algorithms and constraint handling techniques depends on their particular implementations and the constraint solver used. Although there are available tools for some of the test suite generation algorithms studied, they are implemented by different developers with different programming languages, and the substitution of constraint handlers is not supported. In order to avoid the potential bias, we developed a uniform CCAG framework as the reference implementation. As such, all algorithms used in this study are implemented from scratch using the same framework (the same programming language, tools, platform, and development environment). It is possible that slightly different results, especially different computational costs, might be observed by using different programming languages, or constraint solvers. In particular, we acknowledge that the *Solver* technique can run faster with a more efficient constraint solver. Nevertheless, it will not influence the obtained sizes of test suites, because the constraint solver is only used to determine the validity of each value assignment.

Another internal threat to validity derives from the parameter settings of the algorithms (especially for the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

TABLE 9
The number of test models where constraint handler A is significantly superior (+), indistinguishable (=), or significantly inferior (−) to constraint handler B in terms of k-way failure revelation ability.

	A & B	Verify & Solver			Verify & Tolerate			Verify & Replace			Solver & Tolerate			Solver & Replace			Tolerate & Replace		
		+	=	−	+	=	−	+	=	−	+	=	−	+	=	−	+	=	−
$k=3$	AETG	2	33	0	–	–	–	19	16	0	–	–	–	17	18	0	–	–	–
	DDA	2	31	2	–	–	–	6	29	0	–	–	–	8	27	0	–	–	–
	IPO	1	31	3	–	–	–	6	18	11	–	–	–	7	18	10	–	–	–
	PSO	3	31	1	2	33	0	16	19	0	2	32	1	14	21	0	14	21	0
	SA	0	35	0	9	24	2	4	28	3	7	27	1	5	26	4	1	27	7
	TS	0	33	2	5	30	0	4	21	10	6	28	1	5	21	9	3	22	10
$k=4$	AETG	0	32	3	–	–	–	24	11	0	–	–	–	26	9	0	–	–	–
	DDA	1	33	1	–	–	–	6	28	1	–	–	–	10	24	1	–	–	–
	IPO	0	32	3	–	–	–	6	15	14	–	–	–	7	17	11	–	–	–
	PSO	0	35	0	4	31	0	29	5	1	3	32	0	27	8	0	22	13	0
	SA	1	34	0	9	25	1	6	26	3	7	27	1	11	19	5	2	25	8
	TS	0	34	1	3	29	3	5	20	10	5	28	2	6	21	8	3	20	12
$k=5$	AETG	0	35	0	–	–	–	26	9	0	–	–	–	24	11	0	–	–	–
	DDA	1	34	0	–	–	–	8	26	1	–	–	–	8	26	1	–	–	–
	IPO	0	33	2	–	–	–	8	8	19	–	–	–	8	11	16	–	–	–
	PSO	2	32	1	6	28	1	30	5	0	4	30	1	29	6	0	26	9	0
	SA	1	34	0	12	23	0	12	21	2	8	26	1	9	22	4	2	28	5
	TS	3	29	3	3	28	4	6	20	9	6	26	3	7	19	9	6	19	10
$k=6$	AETG	1	32	2	–	–	–	23	12	0	–	–	–	24	10	1	–	–	–
	DDA	0	34	1	–	–	–	8	25	2	–	–	–	4	30	1	–	–	–
	IPO	1	33	1	–	–	–	6	14	15	–	–	–	7	13	15	–	–	–
	PSO	1	34	0	4	30	1	26	9	0	2	32	1	25	10	0	23	12	0
	SA	0	35	0	9	25	1	8	25	2	8	24	3	8	23	4	2	29	4
	TS	2	32	1	6	28	1	5	20	10	7	28	0	5	21	9	2	25	8

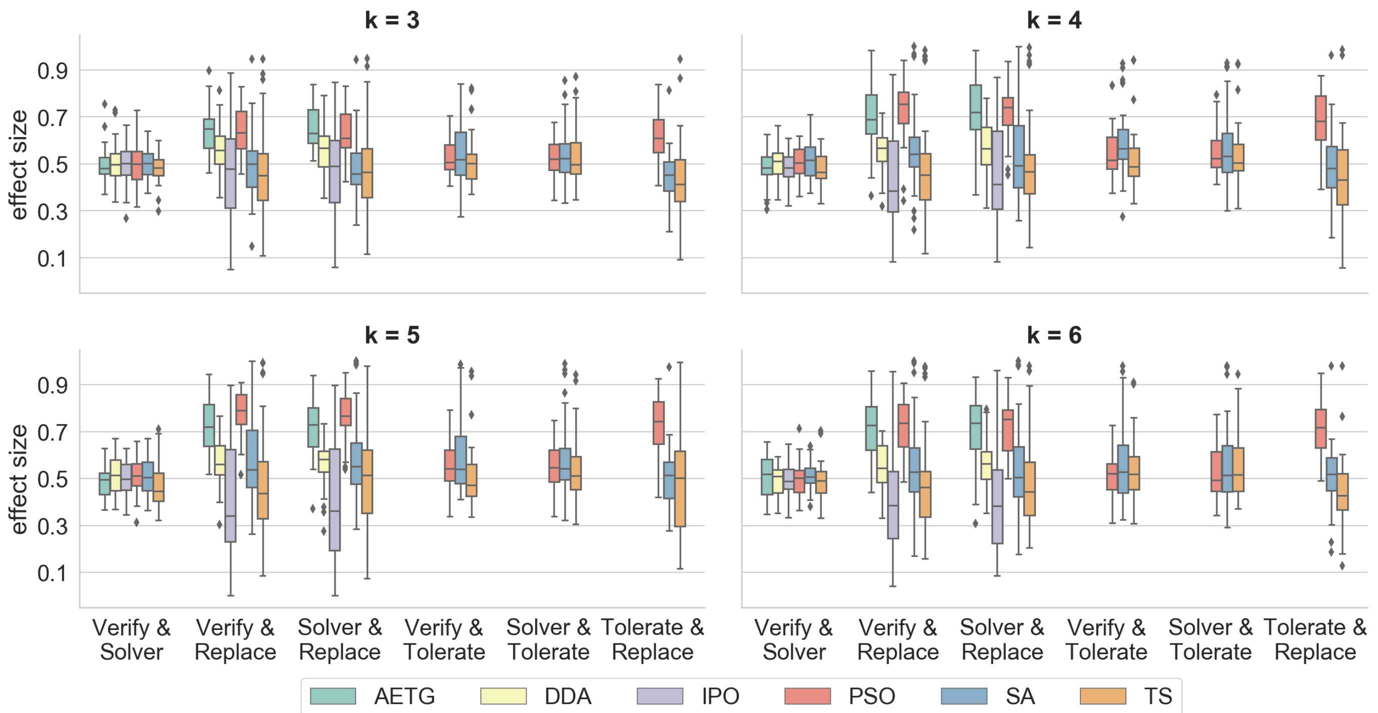

Fig. 4. The distribution of $\widehat{A}_{12}$ statistics for each pair of constraint handling techniques in terms of detecting failures triggered by k parameters.

search-based algorithms). Improved performance might be achieved by exploring the most appropriate settings, but the default values suggested in the literature might be also good enough to assess search-based algorithms [53]. We thus chose to follow the same settings of all algorithms, as reported in their previous studies [22], [23], [24], [25], [26], [27]. While there are two exceptions for SA (number of iterations) and PSO (penalty weight), because their original

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

settings tend to lead to poor performance when the *Tolerate* technique is used.

As far as external threats to validity are concerned, in this study, we only evaluated the performance of algorithms under 35 test models. The algorithms thus may exhibit different behaviours when different subject test models are used. Nevertheless, we have used a well-known benchmark of constrained covering array generation [22], which has been widely used in CT literature [17], [18], [22], [26], [27]. We believe that these test models are representative to investigate the impact of different constraint handling techniques.

6 Conclusion

In this study, we provide a comparative analysis that investigates the impact of four common constraint handling techniques (*Verify*, *Solver*, *Tolerate*, and *Replace*) on six widely used greedy and search-based combinatorial test suite (covering array) generation algorithms (AETG, DDA, IPO, PSO, SA, and TS). Our experimental results reveal that the constraint handler is, indeed, a crucial factor that influences the performance of the test suite generation algorithms into which it is developed.

Specifically, we find that the *Verify* technique implemented with the Minimal Forbidden Tuple (MFT) approach is the fastest choice to handle constraints. The *Replace* technique that resolves constraints as a post-processing phase tends to produce smaller constrained covering arrays than the currently widely used *Verify* and *Solver* techniques, especially for test suite generation algorithms of the one-test-at-a-time framework. Our results also show that it is important to choose a constraint handler that is specifically well-suited to the algorithm and specific goal (test suite size, computational cost, or failure revelation ability). For example, in order to generate the smallest constrained covering arrays, *Replace* is the best choice for AETG, DDA, and PSO; while *Tolerate* could be more promising for SA and TS.

The study we report here could provide insights for the choice of the 'optimal' constraint handler, so that the performance of both existing and newly-designed test suite generation algorithms can be improved. In particular, as a simple implementation of the *Replace* technique is shown to be promising for achieving small test suites, it is desirable to explore that category to develop improved constraint handling technique and generation algorithms. We hope and believe that this paper can offer a better understanding of strengths and weaknesses of constraint handling techniques for CT, so that more studies can be conducted to further improve the area of constrained combinatorial testing.

Acknowledgments

The authors would like to thank Yan Wang and Lejin Wang for their help in implementing the algorithms. This work was partially supported by the National Key Research and Development Plan (No. 2018YFB1003800). This work was also partially supported by the DAASE EPSRC Grant (No. EP/J017515/1) and EPSRC Fellowship (No. EP/P023991/1).

References

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 11:1–11:29, 2011.
[2] R. Mandl, "Orthogonal latin squares: An application of experimental design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.
[3] International Organization for Standardization, "ISO/IEC/IEEE 29119 software testing standard," http://www.softwaretestingstandard.org.
[4] K. Tatsumi, S. Watanabe, Y. Takeuchi, and H. Shimokawa, "Conceptual support for test case design," in *International Computers, Software & Applications Conference*, 1987, pp. 285–290.
[5] K. Tatsumi, "Test case design support system," in *International Conference on Quality Control*, 1987, pp. 615–620.
[6] G. Sherwood, "Effective testing of factor combinations," in *International Conference on Software Testing, Analysis & Review*, 1994, pp. 1–16.
[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.
[8] M. Grindal, J. Offutt, and J. Mellin, "Handling constraints in the input space when using combination strategies for software testing," School of Humanities and Informatics, Tech. Rep. HS-IKI-TR-06-01, 2006.
[9] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *International Symposium on Software Testing and Analysis*, 2007, pp. 129–139.
[10] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, "Constrained interaction testing: A systematic literature study," *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.
[11] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "A survey of constrained combinatorial testing," *CoRR*, vol. abs/1908.02480, 2019. [Online]. Available: https://arxiv.org/abs/1908.02480
[12] H. Nakagawa and T. Tsuchiya, "Towards automatic constraint elicitation in test design: Preliminary evaluation based on collective intelligence," in *International Conference on Automated Software Engineering Workshop*, 2015, pp. 58–61.
[13] R. Tzoref-Brill and S. Maoz, "Syntactic and semantic differencing for combinatorial models of test designs," in *International Conference on Software Engineering*, 2017, pp. 621–631.
[14] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori, "Validation of constraints among configuration parameters using search-based combinatorial interaction testing," in *International Symposium on Search Based Software Engineering*, 2016, pp. 49–63.
[15] A. Gargantini, J. Petke, and M. Radavelli, "Combinatorial interaction testing for automated constraint repair," in *International Workshop on Combinatorial Testing*, 2017, pp. 239–248.
[16] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *International Symposium on Software Reliability Engineering*, 2014, pp. 323–334.
[17] 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.
[18] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation," in *International Conference on Automated Software Engineering*, 2015, pp. 1–12.
[19] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *International Conference on Automated Software Engineering*, 2016, pp. 614–624.
[20] M. Bazargani, J. H. Drake, and E. K. Burke, "Late acceptance hill climbing for constrained covering arrays," in *International Conference on Applications of Evolutionary Computation*, 2018, pp. 778–793.
[21] K. Fogen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *International Workshops on Combinatorial Testing*, 2018, pp. 328–331.
[22] 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.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

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.2019.2955687, IEEE Transactions on Software Engineering

[23] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing, Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[24] Y. Lei, R. N. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[25] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading," *Information and Software Technology*, vol. 86, pp. 20–36, 2017.

[26] 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, 2010.

[27] P. Galinier, S. Kpodjedo, and G. Antoniol, "A penalty-based tabu search for constrained covering arrays," in *Genetic and Evolutionary Computation Conference*, 2017, pp. 1288–1294.

[28] M. B. Cohen and S. Ur, "Combinatorial test design in practice," in *International Conference on Software Engineering*, 2010, pp. 495–496.

[29] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "An empirical comparison of combinatorial testing, random testing and adaptive random testing," *IEEE Transactions on Software Engineering*, 2018, in press, available online.

[30] IEEE Standard Classification for Software Anomalies, *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, 2010.

[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] 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.

[33] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *International Symposium on the Foundations of Software Engineering*, 2013, pp. 26–36.

[34] 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.

[35] A. W. Williams and R. L. Probert, "A practical strategy for testing pair-wise coverage of network interfaces," in *International Conference on Software Reliability Engineering*, 1996, pp. 246–254.

[36] M. Grindal, J. Offutt, and J. Mellin, "Managing conflicts when using combination strategies to test software," in *Australian Software Engineering Conference*, 2007, pp. 255–264.

[37] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *International Workshop on Combinatorial Testing*, 2015, pp. 1–9.

[38] L. Yu, Y. Lei, M. N. Borazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *International Conference on Software Testing, Verification and Validation*, 2013, pp. 242–251.

[39] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149–156, 2004.

[40] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2010.

[41] B. Hnich, S. D. Prestwich, and E. Selensky, "Constraint-based approaches to the covering test problem," in *Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, 2005, pp. 199–219.

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

[43] T. Nanba, T. Tsuchiya, and T. Kikuno, "Using satisfiability solving for pairwise testing in the presence of constraints," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 95, no. 9, pp. 1501–1505, 2012.

[44] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, no. 0, pp. 191–207, 2014.

[45] W. Wang, S. Sampath, Y. Lei, and R. N. Kacker, "An interaction-based test sequence generation approach for testing web application," in *International Conference on High Assurance Systems Engineerng*, 2008, pp. 209–218.

[46] E. Salecker, R. Reicherdt, and S. Glesner, "Calculating prioritized interaction test sets with constraints using binary decision diagrams," in *International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 278–285.

[47] P. Danziger, E. Mendelsohn, L. Moura, and B. Stevens, "Covering arrays avoiding forbidden edges," *Theoretical Computer Science*, vol. 410, no. 52, pp. 5403–5414, 2009.

[48] F. Duan, Y. Lei, L. Yu, R. N. Kacker, and D. R. Kuhn, "Optimizing IPOG's vertical growth with constraints based on hypergraph coloring," in *International Workshop on Combinatorial Testing*, 2017, pp. 181–188.

[49] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *CoRR*, vol. abs/1812.00140, 2019. [Online]. Available: https://arxiv.org/abs/1812.00140

[50] H. Wu and C. Nie, "An overview of search based combinatorial testing," in *International Workshop on Search-Based Software Testing*, 2014, pp. 27–30.

[51] 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.

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

[53] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.