Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection

Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo

Abstract—Combinatorial interaction testing (CIT) is important because it tests the interactions between the many features and parameters that make up the configuration space of software systems. Simulated Annealing (SA) and Greedy Algorithms have been widely used to find CIT test suites. From the literature, there is a widely-held belief that SA is slower, but produces more effective tests suites than Greedy and that SA cannot scale to higher strength coverage. We evaluated both algorithms on seven real-world subjects for the well-studied two-way up to the rarely-studied six-way interaction strengths. Our findings present evidence to challenge this current orthodoxy: real-world constraints allow SA to achieve higher strengths. Furthermore, there was no evidence that Greedy was less effective (in terms of time to fault revelation) compared to SA; the results for the greedy algorithm are actually slightly superior. However, the results are critically dependent on the approach adopted to constraint handling. Moreover, we have also evaluated a genetic algorithm for constrained CIT test suite generation. This is the first time strengths higher than 3 and constraint handling have been used to evaluate GA. Our results show that GA is competitive only for pairwise testing for subjects with a small number of constraints.

Index Terms—Combinatorial interaction testing, prioritisation, empirical studies, software testing

1 Introduction

In this paper we present the results of an empirical study of practical combinatorial interaction testing (CIT). CIT is increasingly important because of the increasing use of configurations as a basis for the deployment of systems [1]. For example, software product lines, operating systems and development environments are all governed by large configuration parameter and feature spaces, for which combinatorial interaction testing has proved a useful technique for uncovering faults.

Two widely used algorithms for CIT are Simulated Annealing (SA) and Greedy. The previous literature assumes a trade-off between computational cost of finding CIT test suites and the fault revealing power of the test suites so-found when time to run the test suites is considered [2], [3], [4]. The 'conventional wisdom' is that Greedy is fast (but its test suites are large and therefore less effective at finding faults quickly). SA, being a meta-heuristic, is generally believed to be slower to compute the test suite, yet it can produce smaller test suites that can find faults faster [4].

The strength of a CIT test suite refers to the level of interactions it tests. In pairwise (or two-way) CIT, only interactions between pairs of configuration choices are tested. As might be expected, there is evidence that testing at higher strengths of interaction can reveal faults left uncovered by lower strengths [5]. We investigate interaction strengths up to six-way, because previous work has shown that there is little value to be gained from higher strengths than this [5].

It is widely believed that SA can only cover the lowest strength (pairwise interaction) in reasonable time; higher strengths, such as those up to five- and six-way feature interactions, have been considered infeasibly expensive, even though they may lead to improved fault revelation [1], [5]. By contrast, though greedy algorithms can scale to such higher strengths [2], it is believed that their results are inferior with respect to test suite size (in general the simulated annealing will produce smaller test suites). This raises two important and related questions that we wish to investigate in this paper:

> *Can we find situations in which meta-heuristic search algorithms such as Simulated Annealing can scale to higher interaction strengths?*

and, if we can find such cases,

> *How well does the Greedy approach compare to Simulated Annealing at higher strengths?*

Until recently, much of the CIT literature has assumed an unconstrained configuration space [1]. This is a questionable assumption because most real-world CIT applications reside in constrained problem domains: some interactions are simply infeasible due to these constraints [6], [7], [8], [9]. Any CIT approach that fails to take account of such constraints may produce many test cases that are either unachievable in practice or which yield expensively misleading results (such as false positives).

- *J. Petke, M. Harman, and S. Yoo are with the Computer Science Department, University College London, London, United Kingdom. E-mail: {j.petke, mark.harman, shin.yoo}@ucl.ac.uk.*
- *M.B. Cohen is with the Computer Science & Engineering Department, University of Nebraska-Lincoln, Lincoln, Nebraska, United States. E-mail: myra@cse.unl.edu.*

Manuscript received 27 May 2014; revised 23 Jan. 2015; accepted 16 Mar. 2015. Date of publication 7 Apr. 2015; date of current version 18 Sept. 2015. Recommended for acceptance by A. Roychoudhury.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2421279

This work is licensed under a Creative Commons Attribution 3.0 License. For more information, see http://creativecommons.org/licenses/by/3.0/

Another type of constraint, often referred to as a soft constraint [6] may also have a role to play. Soft constraints are combinations of options that a tester believes do not need to be tested together (based either on their knowledge of the test subject and/or by a static analysis). Catering for such constraints will not improve test effectiveness, but it may improve efficiency. Unlike other work, we leverage this type of constraint as well in this paper.

Furthermore, practical testers are not so much concerned with finding test suites as with finding an effective prioritisation of a test suite. That is, the order in which the test cases are applied to the system under test is increasingly important, both in general [10] and for CIT [11], [12], [13]. Expecting the tester to simply execute all test cases available is often impractical because it takes too long. It is therefore important that CIT should not merely find a set of test cases, but that it should *prioritise* them so that faults are revealed earlier in the testing process.

Many different meta-heuristic techniques have been developed for CIT, such as simulated annealing [4], genetic algorithms [14], great deluge [15], tabu search [16] and hill climbing [17]. Greedy variants include the Automatic Efficient Test case Generator (AETG) [3], the In Parameter Order General (IPOG) algorithm [2], and the deterministic density algorithm to name a few. Among the state-of-the-art tools are Covering Arrays (CA) by Simulated Annealing (CASA)[1] and Advanced Combinatorial Testing System (ACTS),[2] which uses a greedy algorithm, based on IPOG. Meta-heuristics usually find smaller test suites than a greedy approach, however, the latter is considered to be faster [4]. In this paper we study the Simulated Annealing approach (as implemented in CASA [4]) and the Greedy approach (as implemented in ACTS [18]), because these are two widely used approaches, believed to offer a speed-quality tradeoff and their associated tools are relatively mature. In order to conduct a more thorough investigation we have chosen an additional algorithm for CIT test suite generation in the presence of constraints. We selected one that uses a genetic algorithm. For pairwise testing GA has been shown to be comparable in terms of efficacy and efficiency as SA approaches [14]. However, there have been only a few empirical studies that take into account higher-strength CIT [19] (only three-way, in particular) and none that take constraints into account. We reviewed the available tools [19] and chose the one that provides a method for constraint handling and is able to generate test suites of up to 3.

We present results from empirical studies of these three approaches, reporting on the relationship between their achievement of lower and higher interaction strengths, and their ability to find faults for the constrained prioritised interaction problem, both important practical considerations when applying this technique. There has been little previous work on the relationship between constrained interaction problems and fault revelation, and on the problem of ordering test cases for early fault revelation with respect to constrained higher strength interactions.

1. CASA is available at: cse.unl.edu/~citportal/.

2. ACTS is available by request from: http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html#acts.

This paper addresses this important gap in the literature. We report on constrained, prioritised, CIT using Simulated Annealing, Greedy and Genetic Algorithms approaches for two-way to six-way interaction strengths applied to multiple versions of seven programs from the Software-artifact Infrastructure Repository (SIR) [20]. Our results extend the findings of the previous conference version of paper [21], which considered Simulated Annealing, but neither Greedy nor Genetic Algorithm, for two-way to five-way interaction strengths applied to five of the seven subjects studied here.

The findings of our study provide evidence that challenges some of the assumptions currently prevalent in the CIT literature. Specifically, our primary findings are:

1) We show that higher-strength CIT (even up to six-way interaction strength) is achievable for Simulated Annealing for our real-world subjects, confounding the 'conventional wisdom' that this is infeasible. This apparently surprising result arises because of the role played by constraints in reducing computational effort for simulated annealing.
2) We find that the rate of early fault detection for the greedy algorithm (using 'forbidden tuples' constraint handling where needed for scalability) did not appear to be inferior to that observed for simulated annealing, confounding the belief that there is a tradeoff between the two approaches.

We also establish the following that will aid CIT testers:

1) We show that separate consideration of single- and multi-valued parameters leads to significant runtime improvements for prioritisation and interaction coverage for all three approaches.
2) We show the higher strength CIT is necessary to achieve better fault revelation in prioritised CIT; our empirical study reveals that higher strength CIT reveals more faults than lower strengths. This means that for comprehensive testing, higher strength interaction suites are both desirable (and also feasible, even for Simulated Annealing).
3) We find that lower strength CIT naturally achieves some degree of 'collateral' higher strength coverage, and that it also performs no worse in terms of early fault revelation. This means that we can use lower strength prioritisation as a cheap way to find the first fault.
4) We find that the approach to constraint handling has a pivotal effect on the scalability of the greedy algorithm. Performance can be dramatically improved by using the enumeration of all tuples forbidden by the constraints in place of the default constraint handler. For two subjects studied (GREP and SED), this substitution for the default constraint handler, reduced the execution time from over 8 hours to a mere 8 seconds.
5) We find that the Genetic Algorithm does not scale to higher-strength constrained CIT in terms of execution time, unlike SA and Greedy approaches. Perhaps further development of such global search techniques may further address this issue.

The rest of this paper is organised as follows. We first present Background on combinatorial interaction testing.

Next, we describe our Research Questions and Experimental Setup. Section 5 presents the Results. Future Work and Threats to Validity are described in Sections 6 and 7 respectively. We end with Conclusions containing practical advice for CIT users.

2 Background

In this section we present literature on combinatorial testing as well as notation used throughout the paper.

2.1 Related Work

Combinatorial interaction testing has been used successfully as a system level test method [3], [5], [13], [17], [22], [23], [24], [25], [26]. CIT combines all t-combinations of parameter inputs or configuration options in a systematic way so that we know we have tested a measured subset of input (or configuration) space. Research has shown that we can achieve high fault detection rates given a small set of test cases [3], [5], [13], [25].

Integration of constraint handling into a CIT tool is a non-trivial task. Each constraint can potentially introduce a large number of invalid configurations. Consider, for example, a constraint that disallows two parameters to take particular values. Any extension of such a *disallowed tuple* to other parameters will yield an invalid configuration. Furthermore, combinations of constraints may produce other ones which are not explicitly specified. Grindal et al. [27] evaluated seven constraint handling methods for test case selection. Bryce and Colbourn proposed [6] to avoid disallowed tuples, however, implicit constraints can cause invalid test cases to still be generated. Kuhn et al. [18] implemented a *forbidden tuples* method in their greedy-based ACTS tool [18], where they derive disallowed interactions from the constraints and use these as a validity check for each test case generated. The tool also allows the user to choose a Constraint Satisfaction Problem (CSP) solver to handle constraints. This type of solver has first been used for covering array generation by Hnich et al. [9]. In 2007 a Boolean satisfiability (SAT) solver has been introduced to handle constraints in a greedy [18] as well as an SA-based algorithm [7]. However, only in 2011 improvements introduced by Garvin et al. [4] allowed for efficient constraint handling in an SA-based algorithm for covering array generation.

Many of the current research directions into this technique examine specialised problems such as the addition of constraints between parameter values [6], [7], [24], [27], [28], or re-ordering (prioritising) test suites to improve early coverage [12], [13], [23], [24], [29]. Other work has studied the impact of testing at increasing higher strengths ($t > 2$) [18], [30].

In a recent survey by Nie and Leung [1] CIT research is categorised by a taxonomy to show the areas of study. We have extracted data from this table for three columns, fault detection, constraints and prioritisation. We show this in Table 1 and add a reference to one of the papers from that survey (the survey may include more than one paper per name).

At first glance it might appear from Table 1 that there has been broad coverage of these topics in previous work. However, this is deceptive since most of these CIT aspects are studied in isolation. There are no previous studies that cross the boundaries of prioritisation, constraints and fault detection.

TABLE 1
Overview of Literature on Fault Detection, Prioritisation and Constraints: Extracted from [1]

Authors	Fault detect.	Prioritisation	Constraints
Bryce and Memon [31]	✓	✓	✓
Cohen et al. [3]			✓
Cohen et al. [8]	✓	✓	✓
Grindal et al. [27]			✓
Kuhn et al. [5]	✓		
Nie et el. [32]	✓		
Schroeder et al. [26]			✓

2.2 Preliminaries

In this section we will give a quick overview of the notation used throughout the paper. In particular, a Covering Array is usually represented as follows [33]:

$$CA\left(N; t, v_1^{k_1} v_2^{k_2} \dots v_m^{k_m}\right),$$

where N is the size of the array, t is its strength, sum of $k_1, \dots, k_m$ is the number of parameters and each v_i stands for the number of values for each of the k_i parameters in turn.

Suppose we want to generate a pairwise interaction test suite for an instance with three parameters, where the first and third parameter can take four values and the second one can only take three values. The problem can then be formulated as: $CA(N; 2, 4^1 3^1 4^1)$, which is called *a model* for the CIT problem.

Furthermore, in order to test all combinations one would need $4 * 3 * 4 = 48$ test cases, pairwise coverage reduces this number to 16. Additionally, suppose that we have the following constraints: first, only the first value for the first parameter can be ever combined with values for the other parameters, and second, the last value for the second parameter can never be combined with values for all the other parameters. Introducing such constraints reduces the size of the test suite even further to eight test cases. The importance of constraints is evident even in this small example.[3]

We differentiate between two types of constraints in this work: *hard* and *soft*, terms first proposed by Bryce and Colbourn [6]. Hard constraints exclude dependencies that happen between parameter values. For instance, if turning on 8-bit arithmetic means that we cannot use a division function, then these cannot be tested together. Much of the work on constraints has focused on this type of constraints. Since the challenge is to construct test suites that are guaranteed to avoid these combinations, we cannot have them in our test suites.

Soft constraints, on the other hand, have not, hitherto, received as much attention [6], [7], [8]. These constraints are combinations of parameters that we do not need to test

3. There are, however, cases where a constraint might *increase* test suite size. Consider three parameters $p1$, $p2$ and $p3$, taking values 0, 1 or 2 each. One-way interaction test suite will contain three test cases. If we add a constraint $(p1 \neq 2 \wedge p2 \neq 2) \rightarrow p3 = 2$, we increase one-way interaction test suite size to 4.

together (a tester has decided that combining these parameter values is not needed, but the test will still run if this combination exists).

An example of such a parameter might be combining the string match function in an empty file. While this might be excluded because the tester believes it is unlikely to find a fault, the test case containing this pair still runs.

3 Research Questions

In real-world situations, it is often not feasible to test combinations of the input parameters exhaustively. In these situations, combinatorial interaction testing can help reduce the size of the test suite. Constraints may rule out certain combinations of value-parameters, thereby reducing the size of the test suite even further. The extent of this reduction by constraints motivates our first research question:

RQ1: What is the impact of constraints on the sizes of the models of covering arrays used for CIT?

Most of the literature and practical applications focus on pairwise, and sometimes three-way, interaction coverage. Partially this is due to time inefficiency of the tools available. Kuhn et al. stated in 2008 that "only a handful of tools can generate more complex combinations, such as three-way, four-way, or more (..). The few tools that do generate tests with interaction strengths higher than two-way may require several days to generate tests (..) because the generation process is mathematically complex" [18]. However, recent work in this area shows a promising progress towards higher strength interaction coverage [2], [4], [18]. We want to know how difficult it is to generate test suites that achieve higher-strength interaction coverage when using a state-of-the-art CA generation tool, and the role played by constraints. Thus we ask:

RQ2: How efficient is the generation of higher-strength constrained and unconstrained covering arrays using state-of-the-art tools?

Though the majority of our study is concerned with constrained CIT problems, for which we study the characteristics of both algorithms, we know that the greedy algorithm can potentially scale to handle unconstrained problems, unlike the simulated annealing or genetic algorithm approach. Therefore, we compare the greedy algorithm's execution times on constrained and unconstrained problems (by simply dropping the constraints from our subjects to yield unconstrained versions). This allows us to partially assess the impact of constraints on scalability.

Greedy [2], [3], [18] and meta-heuristic search [4] are the two most frequently used approaches for covering array generation [4]. Both involve a certain degree of randomness. For instance, simulated annealing and genetic algorithm, meta-heuristic search techniques, randomly select a transformation, apply it, and compare the new solution to the previous one to determine which should be retained. Greedy algorithms are less random, yet they nevertheless make random choices to break ties. This motivates our next research question:

RQ3: What is the variance of the sizes of CAs across multiple runs of a CA generation algorithm?

Prioritising test cases according to how many pairs of parameter-value combinations are already covered (i.e., pairwise coverage) has been found to be successful at finding faults quickly [29]. A question arises: "what happens when we prioritise according to a higher-strength coverage criterion?". Note that any t-way interaction also covers some $(t-i)$-way interactions. Thus we want to investigate the relationships between the different types of interaction coverage:

RQ4: What is the coverage rate of k interactions when prioritising by t-way coverage?
- What is the coverage rate of pairwise interactions when prioritising by higher-strength coverage?
- What is the coverage rate of t-way interactions when prioritising by lower-strength coverage?

In other words, we want to know what is the *collateral* coverage of k interactions when prioritising by t-way coverage?

Testers often do not have enough time or resources to execute all test cases from the given test suite, which is why test case prioritisation (TCP) techniques are important [10]. The objective of TCP is to order tests in such a way that maximises the early detection of faults. This motivates our next research question:

RQ5: How effective are the prioritised test suites at detecting faults early?
- Which strength finds all known faults first?
- Which strength provides the fastest rate of fault detection?
- Does prioritising by pairwise interactions lead to faster fault detection rates than when prioritising by higher-strength interactions?
- Is there a 'best' combination when time constraints are considered, for example, creating four-way constrained covering arrays and prioritising by pairwise coverage?

Finally, we would like to know, given the range of approaches available for CIT test suites generation, which is the best one. Our main aim, thus, is to answer the following question:

Which approach should be chosen when generating CIT test suites under constraints?

By answering these research questions, we aim to help the developers and users of CIT tools in their decisions about whether to adopt higher strength CIT.

4 Experimental Setup

In order to answer the questions posed above, we conducted the experiments presented in this section.

4.1 Constrained Testing Models

We have used five C subject programs: FLEX, MAKE, GREP, SED and GZIP, as well as two Java programs: NANOXML and SIENA. Their sizes in Uncommented Lines of Code, measured with `cloc`[4] are presented in Table 2. These are obtained from the Software-artifact Infrastructure Repository [20].

4. http://cloc.sourceforge.net

TABLE 2
Uncommented Lines of Code of Subjects/Versions

Subjects	Ver. 1	Ver. 2	Ver. 3	Ver. 4	Ver. 5	Ver. 6	Ver. 7
FLEX	9,581	10,297	10,319	11,470	10,366	-	-
MAKE	14,459	29,011	30,335	35,583	-	-	-
GREP	9,493	10,017	10,154	10,173	10,102	-	-
SED	5,503	9,884	7,161	7,101	13,419	13,434	14,477
GZIP	4,604	5,092	5,102	5,240	5,754	-	-
NANOXML	1,894	2,532	3,106	-	3,279	-	-
SIENA	2,958	2,959	-	-	3,007	-	-

We chose these C subjects in order to compare our results against the ones obtained previously in the literature (for example, in the work of Qu et al. and Qu and Cohen [13], [30]). Moreover, these five C subjects come with test plans described in the Test Suite Specification Language (TSL) [34]. In this previous work, the modified versions of the subjects were used that did not contain constraints in the input models. NANOXML and SIENA were also chosen to include a subject that was written in another programming language.[5] This set of seven subjects includes all those presently available in the SIR repository for which fault matrices and TSL specifications are available.

We use the TSL description to extract the relevant parameters and values and constraints. In the case of NANOXML the code was divided into several TSL files. We chose the largest one for our experiments.[6] Since TSL is created by a tester, it includes knowledge of the system that combines both hard and soft constraints. TSL contains some single-valued parameters labeled as either *error* or *single*. These are parameters that should be tested alone. We utilize the existence of these single-valued parameters in our experiments. An example is turning on the "help" feature. While this is a hard constraint, turning "statistics" on and off in FLEX would be considered a soft constraint. The test developer has decided that this feature is unlikely to interact with others. We use the constraints from these TSL files without modification for our experiments to mimic what a real user would do (SIR was developed with this goal in mind).

This approach to evaluation also removes bias that from our decision making about which constraints to retain or remove. For the generation of Covering Arrays, we have only considered parameters having at least two possible values.[7] This was to decrease the computation effort of the CA generation tool we used.

We encoded all of the constraints as hard constraints so that they do not appear in our test suite with the aim of reducing the combinatorial space. In the resultant test suite, all single-valued parameters (i.e., parameters that contained only one value that could be combined with other parameters) were simply added to each of the test cases for completion.

4.2 CA Generation

We use CASA, ACTS and a genetic algorithm provided by Shiba et al. [14], which we call GAcit, for the generation of Covering Arrays. CASA is one of the few freely available CA generation tools that can handle logical constraints explicitly specified by the user. It is based on simulated annealing and is known to generate smaller covering arrays than the greedy algorithms for constrained CIT models [4].

Another reason to use CASA is to avoid one potential source of experimental bias. Most of the tools that are based on a greedy algorithm also perform prioritisation during the process of generating the covering array. This occurs because the greedy algorithm always chooses the test case that contains the largest number of uncovered t-tuples. However, since our research questions include investigation of the impact of reduced test suites on the fault detection rate as well as the impact of various prioritisation criteria, we use simulated-annealing, an algorithm that does not implicitly perform prioritisation during its selection phase.

Since greedy approaches are also popular and are considered to be faster than SA-based methods for CA generation, we use the ACTS tool as well for comparison. We also use GAcit to show an alternative method, that has only been proposed for CIT in the last few years.

4.3 CA Prioritisation

After generating t-way covering arrays, we prioritise each of these according to multiple t-way prioritisation criteria (for $2 \leq t \leq 6$). There are standard prioritisation algorithms in the literature: Bryce and Memon [31], and Manchester et al. [35], for example.[8]

For our experiments, we use a variation of the algorithm by Bryce and Memon [31]. We note that this differs from the code-coverage weighted prioritization of Qu et al. [13]. The original algorithm iterates through test cases and retains the one test case that covers the largest number of currently uncovered t-tuples. We also differentiate this from prioritisation during construction (or regeneration) which was used by Qu et al. [24]. In that work new test suites are generated each time.

Note that, in the original algorithm, despite ties being broken at random, the test cases later in the suite have a higher chance of getting picked. Consider the case when all

5. At the time of subject selection, NANOXML was the only Java program that contained TSL specifications as well as respective fault matrices in SIR. SIENA was missing fault matrices which were generated for the purposes of this paper.

6. Exhaustive test suites for all other TSL files contained up to 14 tests each. Thus we only concentrated on generating CIT test suites for the largest TSL file.

7. We note here that some values were immediately prohibited by the constraints. For example, if an 'error' constraint is found, there is no need for checking it's interaction with values for all the other parameters.

8. Note that the two algorithms differ only at the pre-processing stage.

n tests cover the same amount of uncovered t-tuples. The first test will be picked for the current maximum first. However, the probability of it being actually picked as the next test case in our prioritised test suite is 0.5^n, since at each tie breaking point it has to *win* over the next test case. Hence, we gather all test cases whose count of currently uncovered t-tuples is maximal (lines 18-19 in Algorithm 1), and then pick one at random (line 22). Thus each will be picked with probability $1/n$. In order to implement these modifications we add an array, holding all the test suites which cover the same amount of uncovered t-way interactions (line 9). Furthermore, we keep a Boolean mapping from test cases to t-tuples to mark those currently uncovered t-tuples contained by a given test case. We also record the total number of currently uncovered t-tuples contained by a given test case (lines 7, 13). These mappings were updated whenever a new test case was marked as used in order to avoid constantly re-calculating the number of uncovered t-tuples for each test case. The pseudocode for the algorithm used is presented in Algorithm 1.

Algorithm 1. Pseudocode for test suite prioritisation.

1: CA = test suite to prioritize
2: gather all valid t-tuples based on CA
3: $mapping$ = []
4: $sums$ = []
5: **for all** $test$s in CA **do**
6: $mapping[test]$ = [True if t-tuple$_i$ in $test$, else False]
7: $sums[test]$ = sum($mapping[test]$)
8: **end for**
9: $bestTest$ = a test that covers the most unique t-tuples
10: add $bestTest$ to $TestSuite$
11: $selectedTestCount = 1$
12: **while** $selectedTestCount <$ size(CA) **do**
13: update $sums$, $mapping$
14: remove $sums[bestTest]$, $mapping[bestTest]$
15: $tCountMax$ = max($sums$)
16: $bestTests$ = []
17: **for all** $test$s in $sums$ **do**
18: **if** $sums[test]$ == $tCountMax$ **then**
19: add $test$ to $bestTests$
20: **end if**
21: **end for**
22: $bestTest$ = random test from $bestTests$
23: add $bestTest$ to $TestSuite$
24: $selectedTestCount$++
25: **end while**

4.4 Interaction Coverage Metric

To calculate the t-way interaction coverage of a given test suite we use Algorithm 2. We noticed that all of our subjects contain single-valued parameters. Sometimes there are many such single-valued parameters, for example, 69 percent of all the parameters in the case of FLEX. Therefore, we consider these separately. To generate covering arrays, we first exclude single-valued parameters from the models. Once an array is generated, we *extend* each test case with single-valued parameters for completeness. In order to calculate interaction coverage rates efficiently we use CAs output by our tools (i.e., without single-valued parameters). Each test case is evaluated in turn (line 7 in Algorithm 2). For each t-way tuple in the test case, if it's not already covered (line 9), we update the tuple count of covered tuples of length t (line 11). Next, we extend the tuple count of tuples of length $t+1$ and above by considering single-valued parameters (line 13). Finally, to calculate the total number of valid t-way interactions that should occur in the final test suite we use the following combinatorial identity (called Vandermonde's identity):

$$\binom{m+n}{t} = \sum_{i=0}^{t} \binom{m}{i}\binom{n}{t-i},$$

where m and n are the numbers of single- and multi-valued parameters respectively and t is the interaction strength (line 22).

Algorithm 2. Pseudocode for the rate of t-way coverage.

1: CA = a given test suite (without single-valued parameters)
2: $singleparams$ = number of single-valued parameters
3: $multipliers = [\binom{singleparams}{t},]$ number of t-way interactions covered by single-valued parameters
4: $counttuples = copy(multipliers)$ number of t-way interactions covered by the test suite so far
5: $coverage$ = [] number of t-way interactions covered by each test case
6: $tuples$ = [] covered t-tuples
7: **for** j = 1 **to** size(CA) **do**
8: **for all** t-tuples in $test_j$ **do**
9: **if** t-tuple not in $tuples$ **then**
10: add t-tuple to $tuples$
11: $counttuples[t]$+=1
12: **for all** $higherstrength$ **do**
13: $counttuples[higherstrength] +=$ $multipliers[higherstrength - t]$
14: **end for**
15: **end if**
16: **end for**
17: $coverage[test_j]$ = $counttuples$
18: **end for**
19: $multituples$ = all valid t-way tuples for a given model (without single-valued parameters)
20: $singletuples = copy(multipliers)$ all valid t-tuples for single-valued parameters
21: $alltuples$ = [] all t-tuples to be covered
22: $alltuples[t] = \sum_{i=0}^{t} \binom{multituples[t]}{i}\binom{singletuples[t]}{t-i}$
23: $rate = coverage/$number of all valid t-tuples * 100%

To compare how quickly each prioritised test suite achieves the interaction coverage of a specific strength, we define an Average Percentage of Covering-array Coverage (APCC) metric following the Average Percentage of Fault Detection (APFD) metric [36]. Given m interactions to cover and n test cases, let I_i be the index of the first test case that covers the interaction I. APCC is defined as follows:

$$APCC = \left(1 - \frac{\sum_i^m I_i}{nm} + \frac{1}{2n}\right) * 100.$$

APCC measures the area under curve for the plot of increasing interaction coverage for a prioritised test suite. Fig. 1 illustrates the metric using the test suite generated for

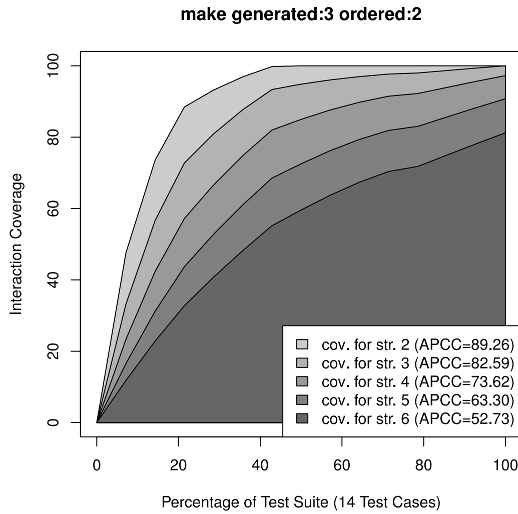


Fig. 1. Interaction coverage of three-way covering array for MAKE prioritised by pairwise coverage (obtained with CASA).

MAKE. It takes 14 test cases (i.e., 100 percent of test cases of the three-way test suite presented) to achieve 100 percent coverage for three-way interaction coverage. The test suite achieves 100 percent coverage for both three-way and pairwise interaction coverage.

4.5 Fault Detection

We measure the fault detection capability of each prioritised test suite. We gather all available software versions of the seven subjects from SIR with seeded faults provided as part of SIR. In order to avoid experimenter bias and ensure repeatability we only used the faults provided with each of the subject tested in SIR. In our study we concentrate on faults, which are detected by full TSL suites and are not triggered by single or error value assignments (marked as such in TSL suites; these need not be tested with any other parameters). Numbers of faults studied for each subject are presented in Table 3. For each of the test suites we gathered the number of faults detected by every i tests. We note that these faults were hand-seeded and do not necessarily represent interaction faults. However, in prior research, these have been shown to be sensitive to the strength of CIT [13], [30].

5 Results

This section presents the results of all the experiments conducted and answers the research questions. We address the first three questions in the next section.

5.1 CA Generation Under Constraints

For FLEX, MAKE and GREP modified TSL descriptions were used by Qu et al. and Qu and Cohen [13], [30] in order to create unconstrained models. We note here that some parameter values were omitted, while some others were combined. The reason for these modifications was to “obtain exhaustive suites that retain close to the original fault detection ability” [13]. Qu et al. also note that “in a real test environment an unconstrained TSL would most likely be prohibitive in size and would not be used” [13]. The sizes of the covering arrays generated for these modified files are presented in Table 4. For FLEX and GREP, the numbers for

TABLE 3
Number of Faults Studied for Each Subject

FLEX	MAKE	GREP	SED	GZIP	NANOXML	SIENA
50	2	12	21	5	16	4

TABLE 4
Covering Array Sizes for Modified Unconstrained CIT Models [13], [30]

CIT Specification	Size $t=2$	Size $t=3$	Size $t=4$	Size $t=5$
FLEX				
$CA(N;t,2^4 3^1 16^1 6^1)$	96	288	NA	NA
GREP				
$CA(N;t,4^1 3^1 2^1 3^1 2^1 12^1 4^1)$	48	192	NA	NA
MAKE				
$CA(N;t,3^1 2^2 5^1 3^2 2^1 4^1)$	20	60	180	540

$t=4$ and $t=5$ were not provided, most probably due to time restrictions of the CA generator used.

5.1.1 Efficiency

The constrained CIT models we use are generated directly from TSL descriptions from SIR and exclude the single-valued parameters. We ran the CASA, ACTS and GAcit tools 20 times on each model on a MacBook Air laptop with an Intel Core i7 processor, running at 1.7 GHz with 8 GB of RAM. Figs. 2 and 3 present the runtime information of generated Covering Arrays. Time limit of 3 hours was set. Note that the runtime variations are much smaller in the case of the ACTS tool. Similarly, almost no variation was recorded for GAcit.

In the case of the genetic algorithm used, with exception of GREP, all pairwise runs where achieved within 3 seconds each. Three-way and higher-strength suites were not generated within 3 hours. Moreover, for SED GAcit was not able to provide a pairwise test suite within 3 hours, while each pairwise test suite generation run for GREP took just under 8,000 seconds.

For both CASA and ACTS most runs took fewer than 20 minutes, as shown in Figs. 2 and 3.[9] If a different setting than the default one was used to generate a CA, then these runs are marked with * or ** in the boxplots. For the three-way criterion of GREP and SED, CASA was terminated after 3 hours: subsequently, we ran CASA again, with the ‘known size’ parameter set to the best result obtained within 3 hours in these two cases. In case of SED and six-way criterion, CASA run into out-of-memory error without producing an initial array. ACTS has not produced a result for SED and GREP and six-way criterion within the 3 hour time limit, thus we changed the default settings to get the covering arrays.

We changed the constraint handling method to ‘forbidden tuples’ and obtained the arrays within 8 seconds each. Such fast runtimes are quite surprising, since this constraint handling method enumerates all forbidden combinations. (Similar method is used in GAcit). The number of

9. In case of FLEX and six-way criterion one run exceeded the 3 hour limit, thus we excluded it from the boxplot.

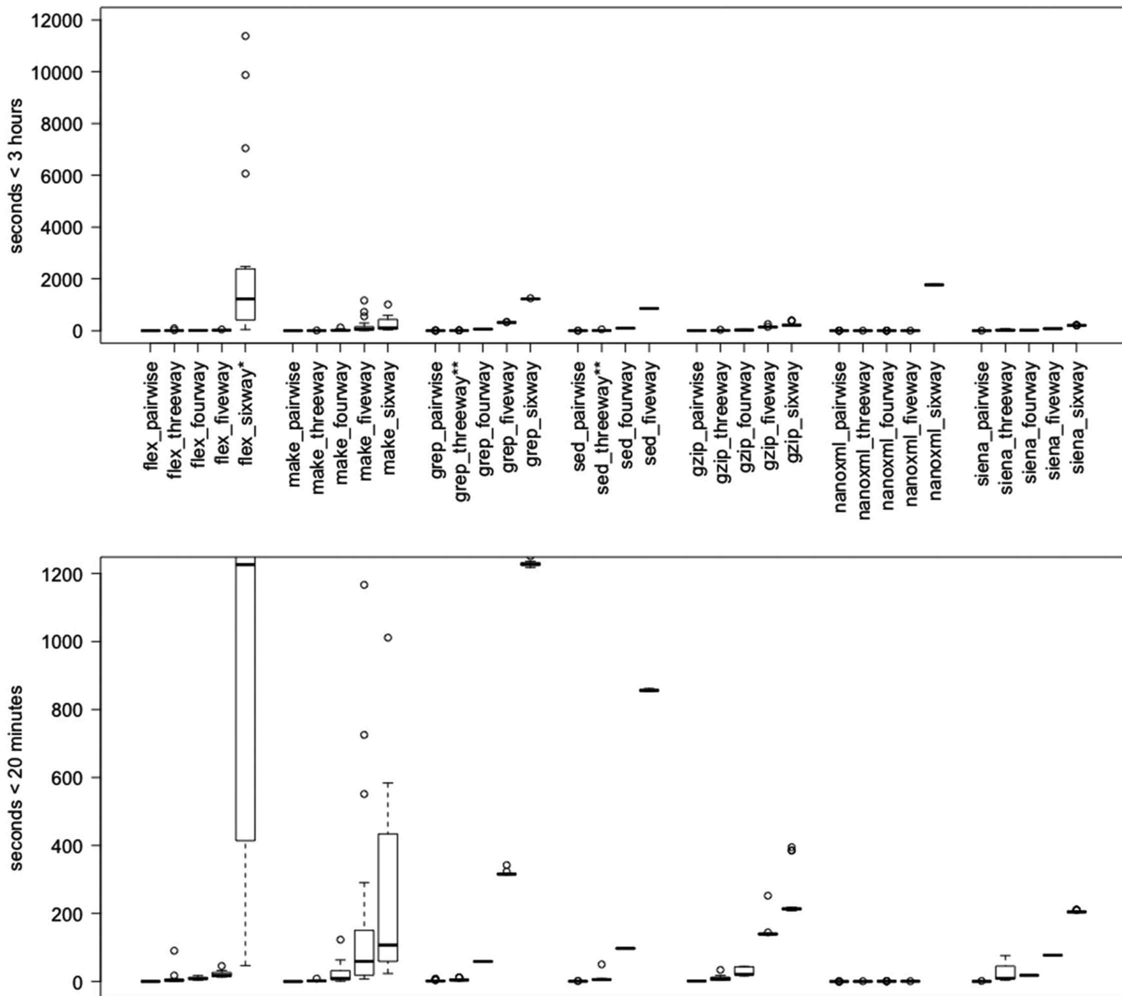


Fig. 2. All CA Generation Runtimes (CASA) with 20 min zoom.

disallowed interactions increases with the number of constraints, making enumeration effort more time consuming and more difficult due to interactions between the constraints. ACTS uses a solver for Constraint Satisfaction Problems as its default for constraint handling. However, the experiments presented in this paper show that this might not be the most efficient method.

5.1.2 Efficacy

The sizes of the smallest constrained CAs we generated using CASA, ACTS and GAcit are presented in Tables 5 and 6. In the case of GREP and SED for $t \geq 4$, only the numbers of unique rows are reported in Table 5. Sizes denoted with * and ** were *not* obtained using default tool settings. The tables also include the number of tests in the original exhaustive TSL test suite from SIR. Tables 4 and 5 provide an answer to *RQ1*: *constraints can reduce the size of CIT models significantly*.

Each test suite among the 20 runs of ACTS had the same size with no variation in test cases. Thus, in contrast to CASA and GAcit (for which maximum size variation was 4), ACTS produces deterministic results. However, ACTS test suite sizes are in almost all cases bigger than in the case of CASA and GAcit, as shown in Table 5. Minimal results from CASA are shown in Table 5, while size variations in Fig. 4. Note that the ACTS tool produced a smaller test suite for GREP and four-way coverage. Since in that case CASA actually produced duplicate test cases (which were later removed), we suspect that an SA algorithm that prevents duplicates would have produced a test suite at least as small as the one generated by ACTS.

Furthermore, minimal test suites in Table 5 differ from the ones presented in our earlier work [21], which poses the question: how many times should CASA be run to obtain the smallest test suite or one whose size is small enough?

We present the CIT models for the original TSL files from SIR with all the constraints and parameter order ignored in Table 7.

Results presented in this section provide strong evidence that constraints play an important part in the efficiency of covering array generation. At the modelling stage, constraints allow for certain values to be excluded from CIT because, for instance, these correspond to error states or cases that do not require further interaction (e.g. printing the 'help' message).

5.1.3 Test Suite Reduction

Excluding single-valued parameters allows further model reduction without compromising the test suite. These can

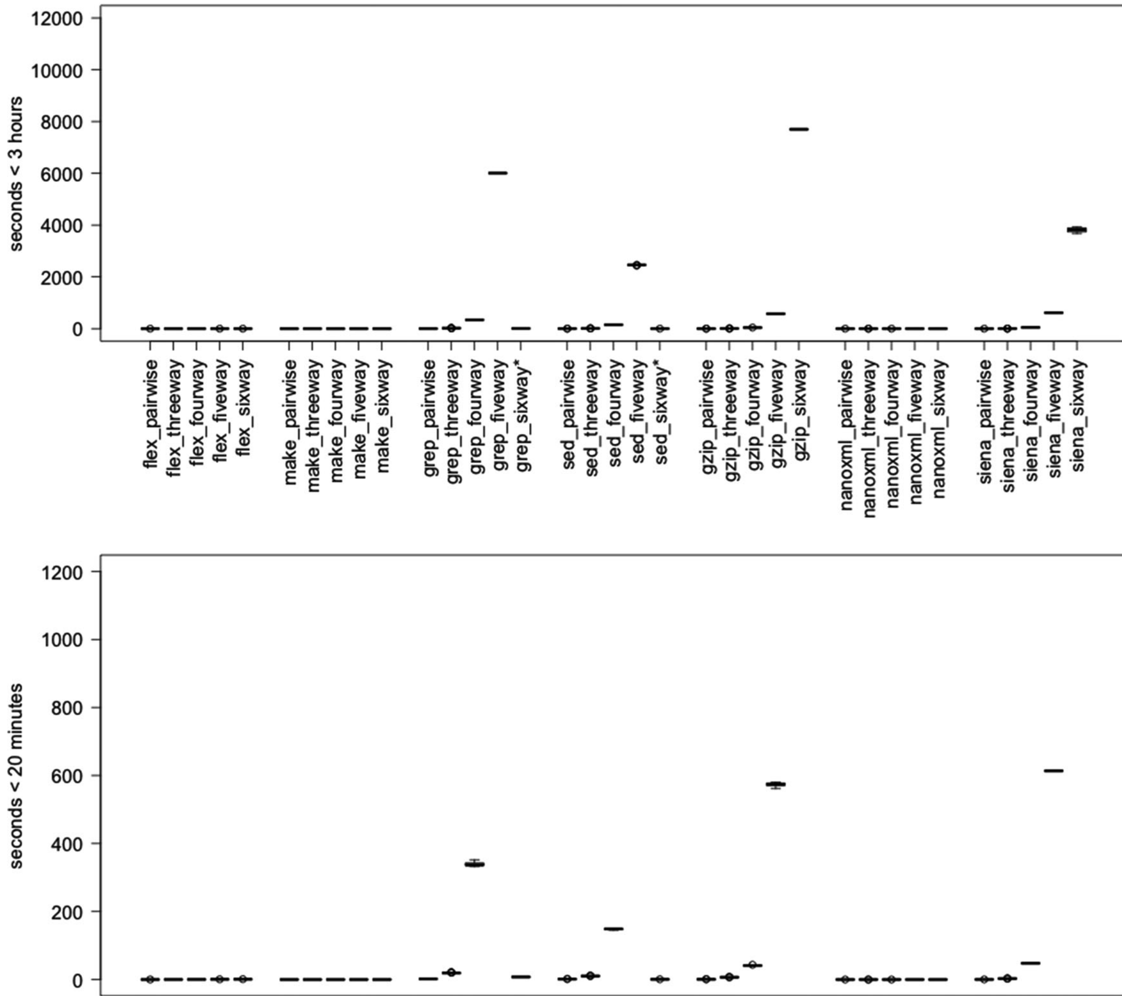


Fig. 3. All CA Generation Runtimes (ACTS) with 20 min zoom.

be added to each row of the CA generated in the post-processing stage, relieving the tool of the need to consider tuples involving such single-valued parameters. This in turn improves CIT test suite generation efficiency. In case of the constrained model for FLEX, for instance, 20 out of 29 parameters are single-valued. There are almost 39 thousand six-way combinations between just the 20 parameters, not to mention six-way tuples that involve also the other nine parameters. Even though these are already covered by the first test case generated, a CIT tool would still count these tuples to establish coverage. We used the ACTS tool to generate a six-way interaction test suite for two FLEX models: one that contained the single-valued parameters and one that didn't. Exclusion of single-valued parameters led to five times speedup. Moreover, the number of tuples covered in the resultant test suites dropped from over 2.5 million to just under eight thousand. Therefore, it is important for CIT testers to consider single- and multi-valued parameters separately in the input model.

The significance of these reductions can be seen in Table 5. The number of test cases generated decreases significantly when compared to the full TSL suite. In the case of MAKE, for instance, five-way coverage is achieved with only 64 tests (using CASA), while the exhaustive test suite contains 793 test cases.

5.1.4 Other Considerations

With regards to the generation effort of CASA, in some cases the variation between runtimes has been significant. This may stem from the different seeds used for the stochastic simulated annealing. At each run, the algorithm starts with a randomly generated solution, which might be either very close to or very far from the actual solution. CASA determines the size of CAs in a stochastic way: it is possible that it gets 'stuck' and works harder on some problems because of a bad starting point. We also note that the biggest runtime variation occurred in the case of MAKE and FLEX (see Fig. 2), which have one of the least constrained CIT models. In fact, there is only one constraint between two (or more) parameters in the TSL model for MAKE. CASA was engineered to work well on constrained CIT models, which might explain this behaviour.

It is also worth noting that in cases where CASA times out it is possible to restart the solver with fixed test suite size. This is not possible for ACTS. Thus even though the latter might generally be faster, when given restrictive time constraints an SA-algorithm might still be a better choice. However, ACTS did not generate a test suite for GREP and SED for interaction strength 6

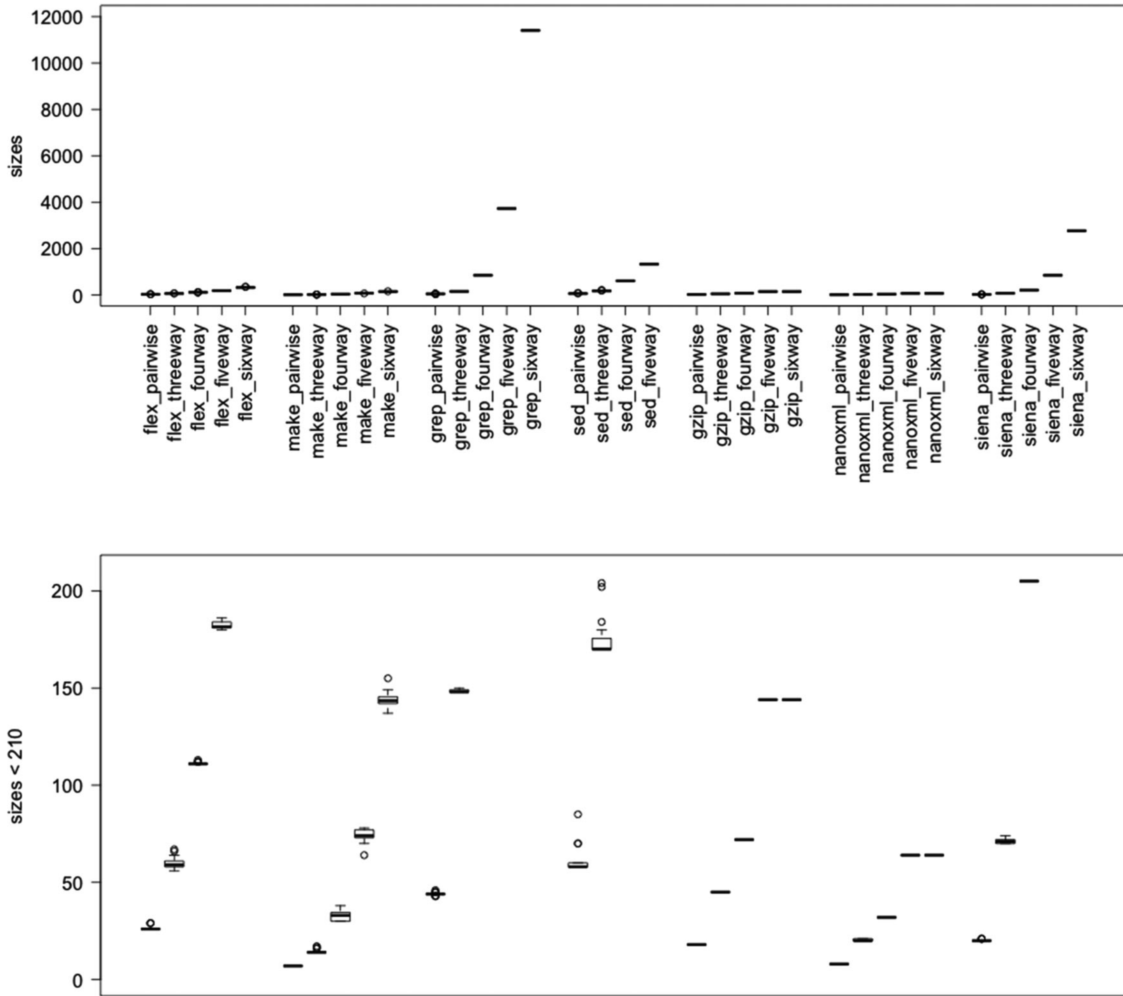


Fig. 4. All CA size variations (CASA) with zoom $<$ 210.

TABLE 5
Constrained Covering Array Sizes for the CASA, ACTS and GAcit Tools (Minimal Out of 20 Runs)

CIT specification	Size $t=2$			Size $t=3$			Size $t=4$			Size $t=5$			Size $t=6$			TSL full
	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	
FLEX $CA(N;t,2^2 3^2 2^4 5^1)$	**26**	27	**26**	**56**	66	-	**111**	130	-	**180**	238	-	**310**	366	-	525
MAKE $CA(N;t,2^{10})$	**7**	9	**7**	**14**	19	-	**30**	43	-	**64**	91	-	**137**	185	-	793
GREP $CA(N;t,3^2 4^1 6^1 8^1 4^1 3^1 2^1 5^1)$	**43**	46	44	**148***	153	-	347′	**298**	-	**436′**	**436**	-	**438′**	**438****	-	470
SED $CA(N;t,2^4 6^1 10^1 2^1 4^1 2^2 3^1)$	**58**	**58**	-	**170***	**170**	-	**324′**	**324**	-	**324′**	**324**	-	Err	**324****	-	360
GZIP $CA(N;t,2^{13} 3^1)$	**18**	21	**18**	**45**	68	-	**72**	108	-	**144**	**144**	-	**144**	**144**	-	214
NANOXML $CA(N;t,2^5 4^1 2^1)$	**8**	**8**	**8**	**20**	24	-	**32**	**32**	-	**64**	**64**	-	**64**	**64**	-	85
SIENA $CA(N;t,3^1 4^1 3^1 5^1 4^2 3^3)$	**20**	22	**20**	**70**	83	-	**205**	216	-	436	**428**	-	518	**516**	-	567

*Best results are marked in bold. ′Denotes sizes after duplicates were removed. *Denotes CAs obtained by first running CASA for 3 hours and then taking the best result as input for the next run. Err denotes out-of-memory error. CAs marked with** were obtained by using the 'forbidden tuples' setting of ACTS for constraint handing. -denotes runs of GAcit which did not produce a valid CA within 3 hours.*

within 3 hours. On the other hand, CASA failed to produce an initial six-way interaction test suite for SED due to an out-of-memory error.

With regards to the generation effort of GAcit, even though it produces comparable sizes to CASA, its the least efficient of the tools used. Given that GREP and SED CIT

TABLE 6
Ratios of Full Test Suite Sizes for Covering Arrays Generated Using the CASA, ACTS and GAcit Tools (Minimal Out of 20 Runs)

CIT specification	Size $t=2$			Size $t=3$			Size $t=4$			Size $t=5$			Size $t=6$			TSL full
	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	CASA	ACTS	GAcit	
FLEX $CA(N;t,2^2 3^2 2^4 5^1)$	**0.05**	0.05	**0.05**	**0.11**	0.13	-	**0.21**	0.25	-	**0.34**	0.45	-	**0.59**	0.70	-	1
MAKE $CA(N;t,2^{10})$	**0.01**	0.01	**0.01**	**0.02**	0.02	-	**0.04**	0.05	-	**0.08**	0.11	-	**0.17**	0.23	-	1
GREP $CA(N;t,3^2 4^1 6^1 8^1 4^1 3^1 2^1 5^1)$	**0.09**	0.10	0.09	**0.31***	0.33	-	0.74′	**0.63**	-	**0.93′**	**0.93**	-	**0.93′**	**0.93****	-	1
SED $CA(N;t,2^4 6^1 10^1 2^1 4^1 2^2 3^1)$	**0.16**	**0.16**	-	**0.47***	**0.47**	-	**0.90′**	**0.90**	-	**0.90′**	**0.90**	-	Err	**0.90****	-	1
GZIP $CA(N;t,2^{13} 3^1)$	**0.08**	0.10	**0.08**	**0.21**	0.32	-	**0.34**	0.50	-	**0.67**	**0.67**	-	**0.67**	**0.67**	-	1
NANOXML $CA(N;t,2^5 4^1 2^1)$	**0.09**	**0.09**	**0.09**	**0.24**	0.28	-	**0.38**	**0.38**	-	**0.75**	**0.75**	-	**0.75**	**0.75**	-	1
SIENA $CA(N;t,3^1 4^1 3^1 5^1 4^2 3^3)$	**0.04**	0.04	**0.04**	**0.12**	0.15	-	**0.36**	0.38	-	0.77	**0.75**	-	0.91	**0.91**	-	1

*Best results are marked in bold. ′denotes sizes after duplicates were removed. *Denotes CAs obtained by first running CASA for 3 hours and then taking the best result as input for the next run. Err denotes out-of-memory error. CAs marked with** were obtained by using the 'forbidden tuples' setting of ACTS for constraint handing. -Denotes runs of GAcit which did not produce a valid CA within 3 hours.*

models contain the largest number of constraints, we might conclude that current GA-based tools have an inefficient way of dealing with constrained CIT. Moreover, generalising to higher-strength also leaves room for improvement.

In conclusion, the state-of-the-art CA generation tools can cope with higher strength CA generation under constraints (*RQ2*). The sizes of generated test suites by ACTS are comparable with CASA producing the best results for strengths five and six for five out of seven subjects studied. Unlike execution time, we observe little variance in CA sizes between the different runs of CASA (Fig. 4), providing an answer to *RQ3*. Furthermore, genetic algorithm's performance is far worse than the two more popular techniques, though competitive for pairwise in five cases. *These observations provide supporting evidence for the best practice, which is to perform a few runs of an SA-based tool with predetermined time-out and then to select the smallest CA generated. If one is using Greedy approach, then one run is enough, however suite size is often bigger than that produced with SA.*

TABLE 7
Constrained and Unconstrained CIT Models

Constrained	Unconstrained
FLEX (32 TSL constraints)	
$CA(N;t,2^6 3^2 5^1)$	$CA(N;t,2^{23} 3^4 5^2)$
9 parameters (30%)	29 parameters (100%)
MAKE (28 TSL constraints)	
$CA(N;t,2^{10})$	$CA(N;t,2^{14} 3^4 4^2 5^1 6^1)$
10 parameters (45′%)	22 parameters (100%)
GREP (58 TSL constraints)	
$CA(N;t,2^1 3^3 4^2 5^1 6^1 8^1)$	$CA(N;t,1^4 2^1 3^3 4^1 5^1 7^1 10^1 13^1 21^1)$
9 parameters (64%)	14 parameters (100%)
SED (58 TSL constraints)	
$CA(N;t,2^7 3^1 4^1 6^1 10^1)$	$CA(N;t,1^1 2^7 3^1 4^3 5^3 6^1 8^2 10^1)$
11 parameters (58%)	19 parameters (100%)
GZIP (69 TSL constraints)	
$CA(N;t,2^{13} 3^1)$	$CA(N;t,1^4 2^8 3^8 4^2 5^1 6^1 34^1)$
14 parameters (56%)	25 parameters (100%)
NANOXML (26 TSL constraints)	
$CA(N;t,2^6 4^1)$	$CA(N;t,1^2 2^{11} 3^6 4^1 6^1)$
7 parameters (33%)	21 parameters (100%)
SIENA (62 TSL constraints)	
$CA(N;t,3^1 4^1 3^1 5^1 4^2 3^3)$	$CA(N;t,4^1 8^1 2^1 8^1 7^1 5^1 8^1 13^1 8^1 9^1 3^1)$
9 parameters (82%)	11 parameters (100%)

5.2 Prioritisation and Interaction Coverage

This section addresses *RQ4*. Following the best practice outlined in Section 5.1, we chose 34 smallest Covering Arrays, out of the CAs we generated using CASA,[10] and 35 Covering Arrays, generated using ACTS, and six Covering Arrays, generated using GAcit[11] for the seven subjects (FLEX, MAKE, GREP, SED, GZIP, NANOXML and SIENA) and for t-way interaction coverage criteria ($2 \leq t \leq 6$). Note that these only contain multi-value parameters (single-valued parameters having been removed). Subsequently, we ordered each of these according to pairwise, three-way, four-way, five-way and six-way coverage using the greedy algorithm presented in Algorithm 1. This produces 375 CAs.

Excluding single-valued parameters also allows significant speed-up for prioritisation. For example, 20 out of 29 parameters for FLEX are single-valued. We report, in Table 8, the runtimes of Algorithm 1 for CIT models of FLEX with and without the single-valued parameters.

Prioritising the same CA according to interaction coverage for different strengths produces significantly different permutations of test cases. Table 9 shows the permutations of the pairwise CA (generated using CASA) of MAKE according to different strength criteria.

Tables 10, 11 and 12 report the interaction coverage achieved by each of the 75 unprioritised CAs. For each CA generated for t-way strength, we measure the interaction coverage for t'-way strength ($2 \leq t' \leq 6$). A t-way strength CA, by definition, achieves 100 percent interaction coverage for strengths lower than t (therefore we omit these criteria from Tables 10, 11 and 12).

10. Since CASA did not produce a result when applied to SED and six-way criterion due to an out-of-memory error.

11. For SED GAcit was not able to provide a pairwise test suite within the 3 hour time limit.

TABLE 8
Runtimes of the Prioritisation Algorithm on Two CIT Models of FLEX

Pairwise CA for FLEX (26 Test Cases)	Prior. Strength	Prior. Time (sec.)
Without single-valued params.	$t=2$	0.036
With single-valued params.	$t=2$	0.213
Without single-valued params.	$t=3$	0.081
With single-valued params.	$t=3$	17.926
Without single-valued params.	$t=4$	0.198
With single-valued params.	$t=4$	986.430
Without single-valued params.	$t=5$	0.248
With single-valued params.	$t=5$	>20 min
Without single-valued params.	$t=6$	0.149
With single-valued params.	$t=6$	>20 min

TABLE 9
Permutations of the Test Suite for MAKE Which Achieves Pairwise Interaction Coverage

MAKE t-way Prioritisation	2-way CA Permutation
$t=2$	$T_3, T_6, T_0, T_1, T_4, T_5, T_2$
$t=3$	$T_5, T_3, T_2, T_0, T_1, T_4, T_6$
$t=4$	$T_3, T_5, T_2, T_0, T_1, T_4, T_6$
$t=5$	$T_6, T_3, T_0, T_1, T_4, T_5, T_2$
$t=6$	$T_0, T_6, T_3, T_1, T_5, T_4, T_2$

For all subject programs but SIENA, pairwise CAs achieve at least 59 percent *collateral* five-way interaction coverage and at least 47 percent *collateral* six-way interaction coverage. This provides an answer to the top level *RQ4*. Note that, for our coverage calculation, single-valued parameters

TABLE 10
Interaction Coverage Results for CAs Generated by CASA

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6
flex	2	26	-	98.59	95.55	91.13	85.68
	3	56	-	-	99.58	98.42	96.37
	4	111	-	-	-	99.94	99.71
	5	180	-	-	-	-	99.99
	6	310	-	-	-	-	-
make	2	7	-	94.37	83.98	71.27	58.33
	3	14	-	-	97.25	90.76	81.22
	4	30	-	-	-	98.90	95.65
	5	64	-	-	-	-	99.65
	6	137	-	-	-	-	-
grep	2	43	-	88.74	74.13	60.03	47.79
	3	148	-	-	97.45	92.38	85.52
	4	347	-	-	-	99.67	98.79
	5	436	-	-	-	-	100.00
	6	438	-	-	-	-	-
siena	2	20	-	76.99	53.41	35.57	23.46
	3	70	-	-	90.40	75.13	58.90
	4	205	-	-	-	97.15	90.54

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6
sed	2	58	-	92.24	82.01	71.75	62.34
	3	170	-	-	98.88	96.56	93.26
	4	324	-	-	-	100.00	100.00
	5	324	-	-	-	-	100.00
	6	-	-	-	-	-	-
gzip	2	18	-	97.56	93.00	87.09	80.51
	3	45	-	-	99.61	98.59	96.86
	4	72	-	-	-	99.95	99.76
	5	144	-	-	-	-	100.00
	6	144	-	-	-	-	-
nanoxml	2	8	-	97.60	92.90	86.62	79.46
	3	20	-	-	99.64	98.63	96.86
	4	32	-	-	-	99.95	99.75
	5	64	-	-	-	-	100.00
	6	64	-	-	-	-	-
	5	436	-	-	-	-	99.59
	6	518	-	-	-	-	-

TABLE 11
Interaction Coverage Results for CAs Generated by ACTS

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6
flex	2	27	-	98.46	95.23	90.61	85.01
	3	66	-	-	99.70	98.84	97.26
	4	130	-	-	-	99.94	99.74
	5	238	-	-	-	-	99.99
	6	366	-	-	-	-	-
make	2	9	-	95.36	86.47	75.14	63.12
	3	19	-	-	98.32	93.83	86.52
	4	43	-	-	-	99.42	97.46
	5	91	-	-	-	-	99.81
	6	185	-	-	-	-	-
grep	2	46	-	89.09	74.86	61.04	48.97
	3	153	-	-	97.40	92.31	85.48
	4	298	-	-	-	99.55	98.32
	5	436	-	-	-	-	100.00
	6	438	-	-	-	-	-
siena	2	22	-	78.03	55.23	37.57	25.22
	3	83	-	-	91.97	78.50	63.41
	4	216	-	-	-	97.54	91.68

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6
sed	2	58	-	92.59	82.61	72.41	62.92
	3	170	-	-	98.94	96.73	93.59
	4	324	-	-	-	100.00	100.00
	5	324	-	-	-	-	100.00
	6	324	-	-	-	-	-
gzip	2	21	-	97.62	93.12	87.25	80.66
	3	68	-	-	99.76	99.11	97.98
	4	108	-	-	-	99.97	99.88
	5	144	-	-	-	-	100.00
	6	144	-	-	-	-	-
nanoxml	2	8	-	97.71	93.21	87.15	80.20
	3	24	-	-	99.69	98.85	97.34
	4	32	-	-	-	99.95	99.75
	5	64	-	-	-	-	100.00
	6	64	-	-	-	-	-
	5	428	-	-	-	-	99.56
	6	516	-	-	-	-	-

TABLE 12
Interaction Coverage Results for CAs Generated by GAcit

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6	Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	6
flex	2	26	-	98.51	95.35	90.79	85.21	gzip	2	18	-	97.59	93.09	87.25	80.74
make	2	7	-	94.56	84.40	71.82	58.90	nanoxml	2	8	-	97.66	93.04	86.82	79.70
grep	2	44	-	88.48	73.72	59.62	47.49	siena	2	20	-	76.59	53.03	35.38	23.42

need to be added back to the CAs in order to produce complete test suites. Interestingly, the lowest rates of coverage for pairwise CAs occurred for MAKE, and GREP and SIENA, which are the least and two of the most constrained subjects, respectively.

To answer the subquestions of *RQ4* on prioritisation, we prioritised each of the 75 CAs according to five different prioritisation criteria (two-, three-, four-, five- and six-way interaction coverage), resulting in 375 prioritised CAs. The results[12] from the prioritisation are aggregated using APCC (defined in Section 4.4) in Tables 13, 14 and 15.

The variation in APCC between the different strengths of prioritisation criteria is observed to have little effect. It caused up to 6.82 percent variation. Strength of covering arrays has no impact on this variation. Thus no prioritisation criterion is clearly better than another. Furthermore, the choice of CA generation tool had little influence on the APCC values. Since CASA produces smaller test suites, one might argue that t-way interactions should be covered more quickly. However, this is not always the case. Covering arrays generated by ACTS achieved up to 8.34 percent faster coverage (in the case of MAKE and three-way generation criterion: 52.72 percent versus 61.06 percent), while CASA achieved up to 3.18 percent faster coverage than ACTS (in the case of NANOXML and pairwise generation criterion: 64.64 percent versus 61.46 percent), as shown in Tables 13 and 14. In 590 cases CAs generated using ACTS produced higher APCC values, while CAs generated using CASA produced higher APCC values in 231 cases. Moreover, since higher-strength covering arrays contain more test cases, they cover more t-way interactions for larger t, as shown in Fig. 5.

This provides answers to the two subquestions in *RQ4*: it seems that there is no clear advantage to be gained by prioritising by interactions of higher/lower strength. Note that whenever the next test case adds a new three-way interaction to the test suite, it does not necessarily mean that a new pairwise interaction has been added.

However, whenever a new two-way interaction is added, then automatically new three-way, four-way, five-way and six-way interactions are covered. Therefore, in terms of interaction coverage, prioritising by the lowest strength (the pairwise interaction criterion) will often prove to be sufficient, as our results confirm. However, a further question arises as to whether the same observation will hold for fault detection rates. This is the question to which we turn in the next section.

12. The complete data and all APCC plots are available at the companion webpage: http://www0.cs.ucl.ac.uk/staff/J.Petke/cittse/html/index.html.

5.3 Fault Detection

This section addresses *RQ5*. Tables 16, 17 and 18 present the percentage of detected faults after 25, 50, 75, and 100 percent of each test suite is executed, aggregated over all versions of subject programs. With FLEX, GREP, SED and NANOXML, CAs with higher generation strength do detect more faults when executed in their entirety. In all cases, the number of faults detected by test cases assigning values to at least two parameters was found to be identical in the case of t-way covering arrays and full TSL test suites provided in SIR.

Thus, we achieve the same fault detection by using a smaller number of tests. For FLEX, this was achieved with four-way covering arrays (obtained with ACTS), for MAKE,

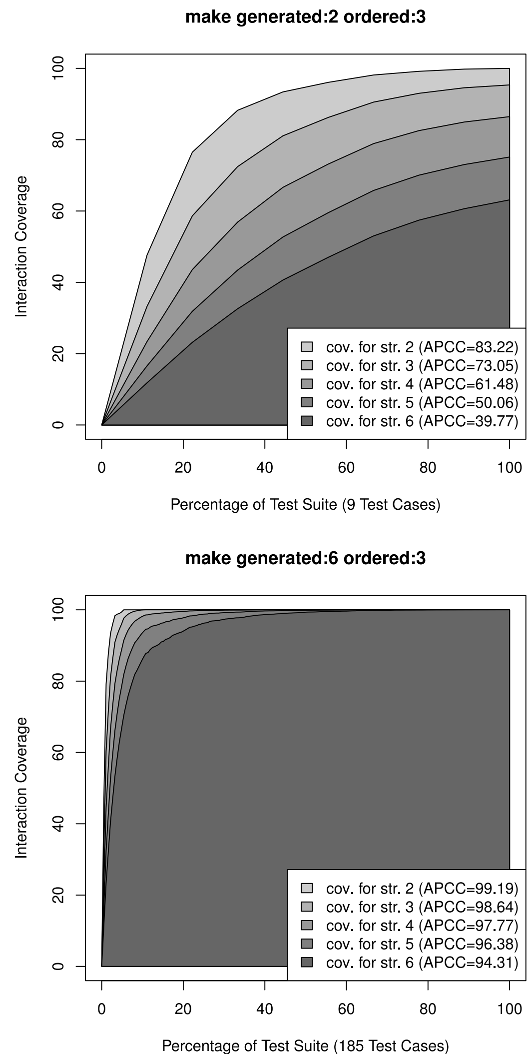


Fig. 5. Comparing APCC for pairwise and six-way CAs for MAKE (generated with ACTS).

TABLE 13
APCC Values for CAs Generated by CASA

Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5	6	Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5	6
	2	2	91.16	85.25	78.51	71.41	64.31		2	2	85.95	72.53	59.79	49.04	40.29
	2	3	90.66	85.02	78.53	71.61	64.61		2	3	85.65	72.47	59.98	49.39	40.73
flex	2	4	90.79	85.01	78.47	71.56	64.60	sed	2	4	85.69	72.63	60.24	49.69	41.00
	2	5	90.10	84.43	78.02	71.22	64.35		2	5	85.26	72.23	59.98	49.55	40.96
	2	6	90.01	84.26	77.79	70.98	64.12		2	6	84.96	72.04	59.82	49.41	40.84
	3	2	95.95	92.78	88.63	83.72	78.27		3	2	95.21	88.34	80.58	73.01	65.97
	3	3	95.70	92.76	89.13	84.86	80.06		3	3	95.15	89.45	82.69	75.66	68.76
flex	3	4	95.53	92.42	88.68	84.39	79.63	sed	3	4	94.97	89.35	82.70	75.75	68.91
	3	5	94.57	91.46	87.86	83.74	79.16		3	5	94.96	89.37	82.80	75.93	69.14
	3	6	94.46	91.14	87.38	83.17	78.56		3	6	94.80	89.17	82.57	75.69	68.93
	4	2	97.94	96.28	93.99	91.16	87.85		4	2	97.51	93.29	88.35	83.39	78.69
	4	3	97.91	96.44	94.49	92.03	89.07		4	3	97.46	94.57	90.83	86.62	82.21
flex	4	4	97.71	96.13	94.16	91.77	88.97	sed	4	4	97.43	94.56	90.97	87.00	82.87
	4	5	97.37	95.67	93.60	91.17	88.37		4	5	97.39	94.50	90.91	86.95	82.83
	4	6	97.08	95.17	92.95	90.43	87.60		4	6	97.28	94.38	90.82	86.90	82.83
	5	2	98.74	97.71	96.27	94.43	92.24		5	2	97.51	93.45	88.55	83.57	78.78
	5	3	98.71	97.83	96.63	95.05	93.09		5	3	97.49	94.57	90.80	86.55	82.10
flex	5	4	98.54	97.61	96.42	94.95	93.16	sed	5	4	97.42	94.55	90.94	86.93	82.77
	5	5	98.45	97.43	96.17	94.64	92.82		5	5	97.37	94.49	90.90	86.94	82.82
	5	6	98.24	97.08	95.66	93.99	92.08		5	6	97.23	94.37	90.83	86.93	82.87
	6	2	99.28	98.63	97.69	96.46	94.97		6	2	0.00	0.00	0.00	0.00	0.00
	6	3	99.18	98.67	97.98	97.05	95.87		6	3	0.00	0.00	0.00	0.00	0.00
flex	6	4	99.16	98.62	97.93	97.09	96.05	sed	6	4	0.00	0.00	0.00	0.00	0.00
	6	5	99.10	98.51	97.78	96.91	95.88		6	5	0.00	0.00	0.00	0.00	0.00
	6	6	98.90	98.23	97.44	96.53	95.47		6	6	0.00	0.00	0.00	0.00	0.00
	2	2	78.72	66.97	54.77	43.49	33.81		2	2	83.44	75.34	67.36	59.85	52.93
	2	3	78.51	67.40	55.48	44.20	34.39		2	3	83.45	75.38	67.45	59.97	53.09
make	2	4	78.51	67.40	55.48	44.20	34.39	gzip	2	4	83.42	75.34	67.43	59.97	53.12
	2	5	78.72	66.97	54.77	43.49	33.81		2	5	82.81	74.85	67.08	59.75	52.99
	2	6	76.95	65.66	53.91	42.97	33.50		2	6	82.22	74.40	66.77	59.56	52.88
	3	2	89.26	82.59	73.62	63.30	52.73		3	2	93.06	89.01	84.42	79.49	74.41
	3	3	89.24	82.78	74.07	63.92	53.41		3	3	93.07	89.07	84.56	79.73	74.73
make	3	4	88.55	81.68	72.94	62.98	52.72	gzip	3	4	93.10	89.08	84.59	79.80	74.85
	3	5	88.55	81.67	72.91	62.94	52.69		3	5	93.02	89.01	84.52	79.74	74.81
	3	6	87.70	81.14	72.69	62.91	52.75		3	6	93.09	89.08	84.59	79.79	74.85
	4	2	94.89	91.11	85.82	78.98	70.88		4	2	95.86	93.23	90.02	86.43	82.61
	4	3	94.94	91.55	86.70	80.15	72.13		4	3	95.86	93.36	90.34	86.95	83.31
make	4	4	94.97	91.74	87.04	80.66	72.75	gzip	4	4	95.85	93.33	90.30	86.89	83.25
	4	5	94.64	91.46	86.82	80.49	72.63		4	5	95.86	93.35	90.32	86.92	83.28
	4	6	94.46	91.26	86.63	80.32	72.51		4	6	95.53	93.00	90.01	86.68	83.11
	5	2	97.65	95.79	92.90	88.92	83.83		5	2	97.93	96.45	94.49	92.20	89.68
	5	3	97.65	96.12	93.73	90.20	85.40		5	3	97.93	96.68	95.14	93.35	91.34
make	5	4	97.65	96.13	93.87	90.55	85.93	gzip	5	4	97.93	96.69	95.17	93.44	91.53
	5	5	97.62	96.05	93.67	90.29	85.71		5	5	97.93	96.68	95.15	93.39	91.46
	5	6	97.44	95.83	93.49	90.17	85.64		5	6	97.93	96.68	95.16	93.41	91.49
	6	2	98.91	98.05	96.64	94.56	91.74		6	2	97.93	96.51	94.67	92.49	90.10
	6	3	98.91	98.18	96.97	95.07	92.32		6	3	97.93	96.69	95.15	93.34	91.30
make	6	4	98.91	98.21	97.14	95.50	93.10	gzip	6	4	97.93	96.69	95.18	93.45	91.55
	6	5	98.87	98.16	97.09	95.50	93.21		6	5	97.93	96.68	95.16	93.42	91.49
	6	6	98.82	98.12	97.06	95.48	93.22		6	6	97.93	96.68	95.15	93.39	91.45
	2	2	80.16	63.43	48.53	36.68	27.64		2	2	80.95	73.22	65.29	57.60	50.41
	2	3	80.01	63.62	48.86	36.98	27.85		2	3	80.95	73.23	65.31	57.63	50.46
grep	2	4	79.43	63.13	48.51	36.77	27.76	nanoxml	2	4	79.52	72.00	64.36	56.94	49.97
	2	5	79.37	62.97	48.40	36.72	27.75		2	5	79.98	72.38	64.64	57.14	50.11
	2	6	78.09	62.47	48.30	36.77	27.83		2	6	76.56	68.39	60.65	53.47	46.93
	3	2	94.29	86.00	76.01	65.92	56.45		3	2	92.11	88.18	83.54	78.42	73.05
	3	3	94.09	87.28	78.50	68.93	59.44		3	3	91.92	88.13	83.72	78.85	73.68
grep	3	4	93.65	86.87	78.24	68.85	59.52	nanoxml	3	4	90.90	87.05	82.75	78.08	73.13
	3	5	93.33	86.60	78.09	68.84	59.62		3	5	91.28	87.14	82.56	77.68	72.62
	3	6	92.91	86.20	77.79	68.64	59.52		3	6	86.68	81.74	76.90	72.11	67.38
	4	2	97.59	93.32	87.71	81.56	75.30		4	2	95.26	92.74	89.48	85.67	81.52
	4	3	97.49	94.53	90.19	84.82	78.84		4	3	95.21	92.66	89.64	86.21	82.47
grep	4	4	97.29	94.45	90.53	85.73	80.28	nanoxml	4	4	95.01	92.59	89.63	86.26	82.57
	4	5	97.24	94.32	90.42	85.73	80.42		4	5	94.57	92.02	89.06	85.74	82.16
	4	6	96.92	94.00	90.16	85.54	80.32		4	6	92.60	89.07	85.42	81.74	78.04
	5	2	98.11	94.51	89.81	84.70	79.54		5	2	97.59	96.18	94.25	91.91	89.28
	5	3	98.00	95.73	92.24	87.83	82.86		5	3	97.60	96.45	95.00	93.24	91.19
grep	5	4	97.87	95.62	92.53	88.73	84.35	nanoxml	5	4	97.44	96.26	94.85	93.22	91.37
	5	5	97.72	95.45	92.39	88.68	84.44		5	5	97.47	96.28	94.83	93.15	91.25
	5	6	97.10	94.85	91.89	88.30	84.19		5	6	96.95	95.30	93.50	91.61	89.66
	6	2	98.09	94.41	89.66	84.56	79.45		6	2	97.60	96.26	94.47	92.30	89.85
	6	3	98.04	95.77	92.26	87.82	82.83		6	3	97.60	96.45	95.01	93.27	91.25
grep	6	4	97.92	95.64	92.57	88.79	84.43	nanoxml	6	4	97.53	96.36	94.91	93.22	91.31
	6	5	97.74	95.41	92.37	88.69	84.50		6	5	97.53	96.32	94.84	93.11	91.19
	6	6	97.50	95.20	92.17	88.53	84.38		6	6	96.74	95.29	93.72	92.03	90.22
	2	2	70.74	47.64	30.42	19.23	12.28		4	5	96.82	92.35	85.28	75.69	64.55
	2	3	70.48	47.66	30.51	19.32	12.34		4	6	96.48	91.81	84.81	75.47	64.57
siena	2	4	69.82	47.33	30.44	19.33	12.36		5	2	98.69	94.27	88.62	82.39	75.69
	2	5	69.26	47.07	30.35	19.31	12.36		5	3	98.65	96.71	92.27	85.89	78.39
	2	6	68.45	46.63	30.19	19.26	12.35	siena	5	4	98.55	96.61	93.26	87.98	80.97
	3	2	91.65	78.03	61.65	46.37	33.81		5	5	98.33	96.38	93.10	88.22	81.82
	3	3	91.21	79.41	63.64	48.07	34.96		5	6	98.01	95.99	92.72	88.02	81.87
siena	3	4	90.48	78.94	63.57	48.23	35.19		6	2	98.93	95.27	90.34	84.78	78.83
	3	5	89.74	78.25	63.18	48.15	35.26		6	3	98.83	97.21	93.36	87.80	81.28
	3	6	89.63	78.05	63.03	48.07	35.25	siena	6	4	98.81	97.17	94.34	89.89	83.93
	4	2	97.28	90.67	81.79	71.49	60.58		6	5	98.64	96.93	94.17	90.12	84.73
	4	3	97.09	92.83	84.79	74.11	62.40		6	6	98.48	96.69	93.92	89.96	84.78
siena	4	4	97.04	92.71	85.52	75.67	64.29								

TABLE 14
APCC Values for CAs Generated by ACTS

Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5	6
	2	2	91.54	85.59	78.72	71.46	64.20
	2	3	91.14	85.51	78.98	72.01	64.95
flex	2	4	91.15	85.44	78.85	71.86	64.82
	2	5	91.00	85.18	78.55	71.56	64.55
	2	6	90.44	84.72	78.21	71.32	64.39
	3	2	96.62	93.96	90.37	86.02	81.12
	3	3	96.51	94.04	90.87	87.05	82.67
flex	3	4	96.16	93.56	90.35	86.56	82.27
	3	5	95.88	93.17	89.90	86.12	81.87
	3	6	95.35	92.59	89.33	85.57	81.37
	4	2	98.27	96.81	94.74	92.14	89.08
	4	3	98.22	96.95	95.29	93.17	90.59
flex	4	4	98.10	96.76	95.07	93.01	90.55
	4	5	97.83	96.34	94.55	92.46	90.03
	4	6	97.55	95.94	94.07	91.91	89.46
	5	2	99.01	98.17	96.95	95.39	93.51
	5	3	99.01	98.35	97.44	96.24	94.72
flex	5	4	98.92	98.21	97.31	96.20	94.85
	5	5	98.79	97.99	97.02	95.88	94.54
	5	6	98.66	97.76	96.70	95.49	94.11
	6	2	99.38	98.84	98.03	96.98	95.69
	6	3	99.37	98.93	98.33	97.54	96.53
flex	6	4	99.27	98.82	98.25	97.54	96.66
	6	5	99.12	98.60	97.99	97.26	96.41
	6	6	99.03	98.45	97.79	97.02	96.14
	2	2	83.24	73.06	61.46	50.02	39.72
	2	3	83.22	73.05	61.48	50.06	39.77
make	2	4	82.88	72.48	60.84	49.47	39.29
	2	5	82.07	72.14	60.86	49.68	39.56
	2	6	82.07	72.12	60.81	49.62	39.49
	3	2	92.02	86.66	79.17	70.07	60.20
	3	3	92.12	86.94	79.74	70.85	61.05
make	3	4	92.14	86.97	79.75	70.86	61.06
	3	5	91.68	86.55	79.33	70.43	60.66
	3	6	90.87	85.79	78.83	70.22	60.65
	4	2	96.49	93.81	89.67	84.05	77.11
	4	3	96.51	94.20	90.64	85.54	78.90
make	4	4	96.46	94.18	90.75	85.85	79.44
	4	5	96.39	94.16	90.76	85.86	79.43
	4	6	95.88	93.62	90.30	85.54	79.25
	5	2	98.37	97.12	95.17	92.33	88.46
	5	3	98.37	97.26	95.49	92.81	89.01
make	5	4	98.36	97.29	95.67	93.23	89.75
	5	5	98.30	97.25	95.65	93.26	89.88
	5	6	98.21	97.16	95.56	93.18	89.81
	6	2	99.20	98.56	97.51	95.95	93.76
	6	3	99.19	98.64	97.77	96.38	94.31
make	6	4	99.19	98.68	97.90	96.68	94.84
	6	5	99.17	98.67	97.89	96.72	95.00
	6	6	99.07	98.56	97.80	96.64	94.95
	2	2	81.56	65.14	50.20	38.14	28.88
	2	3	81.33	65.07	50.27	38.29	29.04
grep	2	4	80.19	64.51	50.12	38.34	29.16
	2	5	79.04	63.78	49.67	38.07	29.00
	2	6	79.56	64.15	49.93	38.25	29.11
	3	2	94.48	86.39	76.50	66.45	57.03
	3	3	94.34	87.50	78.68	69.15	59.75
grep	3	4	94.04	87.41	78.86	69.52	60.22
	3	5	93.80	87.11	78.58	69.30	60.07
	3	6	92.47	86.18	78.02	69.02	59.97
	4	2	97.16	92.72	86.90	80.45	73.79
	4	3	97.06	93.66	88.75	82.79	76.22
grep	4	4	96.89	93.45	88.89	83.49	77.47
	4	5	96.45	93.18	88.75	83.45	77.52
	4	6	96.49	92.99	88.47	83.17	77.30
	5	2	98.05	94.76	90.36	85.44	80.34
	5	3	98.00	95.70	92.16	87.69	82.67
grep	5	4	97.85	95.58	92.48	88.67	84.30
	5	5	97.65	95.36	92.31	88.60	84.36
	5	6	97.47	95.15	92.11	88.44	84.26
	6	2	98.09	94.60	90.05	85.08	80.03
	6	3	97.96	95.73	92.24	87.80	82.81
grep	6	4	97.87	95.63	92.56	88.75	84.37
	6	5	97.66	95.38	92.36	88.70	84.51
	6	6	97.26	94.99	92.03	88.46	84.37
	2	2	74.17	50.44	32.44	20.67	13.30
	2	3	73.91	50.62	32.72	20.88	13.43
siena	2	4	73.43	50.41	32.69	20.92	13.47
	2	5	72.84	50.21	32.64	20.91	13.47
	2	6	70.75	49.12	32.24	20.80	13.45
	3	2	92.92	80.77	65.37	50.40	37.64
	3	3	92.69	82.29	67.37	51.98	38.59
siena	3	4	92.32	81.95	67.41	52.34	39.05
	3	5	91.69	81.35	67.13	52.36	39.24
	3	6	90.64	80.63	66.72	52.18	39.21
	4	2	97.35	90.55	81.54	71.39	60.87
	4	3	97.23	93.22	85.52	75.18	63.76
siena	4	4	97.08	93.08	86.22	76.70	65.62

Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5	6
	2	2	85.65	72.18	59.54	48.87	40.17
	2	3	85.26	72.22	59.92	49.42	40.76
sed	2	4	85.05	72.14	59.96	49.53	40.90
	2	5	84.62	71.70	59.63	49.35	40.84
	2	6	84.53	71.61	59.54	49.26	40.77
	3	2	95.26	88.57	80.92	73.36	66.29
	3	3	95.23	89.46	82.58	75.43	68.44
sed	3	4	94.94	89.28	82.57	75.57	68.70
	3	5	94.81	89.12	82.44	75.49	68.66
	3	6	94.83	89.11	82.41	75.49	68.70
	4	2	97.53	93.23	88.22	83.26	78.58
	4	3	97.48	94.60	90.87	86.66	82.27
sed	4	4	97.45	94.56	90.94	86.94	82.77
	4	5	97.36	94.49	90.93	86.99	82.88
	4	6	97.28	94.40	90.85	86.94	82.87
	5	2	97.54	93.13	88.06	83.08	78.43
	5	3	97.47	94.58	90.82	86.57	82.13
sed	5	4	97.46	94.58	90.98	87.00	82.85
	5	5	97.36	94.49	90.92	86.98	82.88
	5	6	97.34	94.45	90.88	86.94	82.85
	6	2	97.51	93.06	87.91	82.85	78.13
	6	3	97.51	94.60	90.85	86.64	82.25
sed	6	4	97.42	94.55	90.96	86.98	82.83
	6	5	97.39	94.49	90.89	86.92	82.80
	6	6	97.30	94.39	90.84	86.92	82.86
	2	2	85.40	77.58	69.60	61.90	54.75
	2	3	83.89	76.02	68.14	60.60	53.62
gzip	2	4	84.27	76.87	69.29	61.90	54.94
	2	5	84.17	76.73	69.15	61.78	54.86
	2	6	83.60	76.12	68.61	61.36	54.58
	3	2	95.61	92.74	89.15	85.07	80.71
	3	3	95.58	92.99	89.87	86.31	82.44
gzip	3	4	95.61	92.95	89.71	86.04	82.06
	3	5	95.60	93.00	89.88	86.32	82.45
	3	6	95.60	92.95	89.76	86.13	82.20
	4	2	97.24	95.38	92.99	90.20	87.15
	4	3	97.24	95.58	93.54	91.16	88.51
gzip	4	4	97.24	95.57	93.53	91.19	88.61
	4	5	97.24	95.59	93.60	91.31	88.78
	4	6	97.24	95.57	93.54	91.22	88.66
	5	2	97.93	96.51	94.65	92.47	90.07
	5	3	97.93	96.69	95.15	93.33	91.27
gzip	5	4	97.93	96.69	95.19	93.48	91.59
	5	5	97.93	96.67	95.14	93.37	91.42
	5	6	97.92	96.68	95.16	93.40	91.46
	6	2	97.93	96.48	94.58	92.35	89.90
	6	3	97.93	96.69	95.14	93.33	91.29
gzip	6	4	97.93	96.68	95.17	93.45	91.54
	6	5	97.93	96.68	95.15	93.39	91.46
	6	6	97.93	96.68	95.16	93.42	91.49
	2	2	80.64	72.70	64.70	57.03	49.94
	2	3	78.80	71.59	64.25	57.04	50.21
nanoxml	2	4	78.80	71.59	64.25	57.04	50.21
	2	5	77.53	69.30	61.46	54.19	47.58
	2	6	77.74	69.69	61.94	54.69	48.05
	3	2	93.46	90.24	86.36	81.95	77.18
	3	3	93.37	90.20	86.50	82.31	77.74
nanoxml	3	4	93.22	90.05	86.34	82.16	77.62
	3	5	92.72	89.42	85.65	81.48	76.98
	3	6	91.83	88.10	84.01	79.63	75.04
	4	2	95.24	92.69	89.52	85.90	81.98
	4	3	95.05	92.68	89.77	86.39	82.64
nanoxml	4	4	94.85	92.44	89.55	86.25	82.63
	4	5	94.51	91.74	88.59	85.14	81.50
	4	6	92.60	89.45	86.08	82.54	78.90
	5	2	97.61	96.33	94.63	92.56	90.18
	5	3	97.57	96.44	94.98	93.16	91.03
nanoxml	5	4	97.42	96.27	94.87	93.24	91.41
	5	5	97.45	96.22	94.75	93.05	91.15
	5	6	96.29	94.83	93.30	91.68	89.94
	6	2	97.59	96.31	94.61	92.55	90.18
	6	3	97.57	96.44	95.00	93.25	91.22
nanoxml	6	4	97.41	96.27	94.88	93.26	91.43
	6	5	97.06	95.88	94.51	92.92	91.10
	6	6	96.81	95.29	93.67	91.95	90.14
siena	4	5	96.87	92.76	86.03	76.78	65.93
	4	6	96.52	92.05	85.41	76.51	66.02
	5	2	98.64	94.59	89.09	82.80	75.90
	5	3	98.65	96.64	92.28	86.01	78.54
siena	5	4	98.56	96.59	93.19	87.78	80.57
	5	5	98.41	96.40	93.03	88.04	81.51
	5	6	98.10	95.99	92.59	87.70	81.33
	6	2	98.91	94.90	89.73	84.12	78.22
	6	3	98.87	97.22	93.36	87.78	81.22
siena	6	4	98.80	97.15	94.35	89.89	83.90
	6	5	98.66	96.99	94.21	90.12	84.73
	6	6	98.38	96.62	93.86	89.90	84.69

TABLE 15
APCC Values for CAs Generated by GAcit

Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength					Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength				
			2	3	4	5	6				2	3	4	5	6
flex	2	2	91.41	85.65	78.94	71.79	64.60	gzip	2	2	83.44	75.34	67.39	59.89	52.99
	2	3	91.44	85.66	79.00	71.94	64.82		2	3	83.37	75.41	67.59	60.20	53.38
	2	4	90.92	85.20	78.62	71.64	64.61		2	4	83.37	75.41	67.59	60.20	53.38
	2	5	89.49	83.88	77.54	70.81	64.00		2	5	83.41	75.33	67.44	60.02	53.20
	2	6	89.60	83.96	77.56	70.78	63.93		2	6	83.41	75.35	67.46	60.04	53.22
make	2	2	78.39	67.22	55.31	44.07	34.30	nanoxml	2	2	80.58	73.03	65.26	57.68	50.56
	2	3	78.72	67.36	55.31	44.02	34.23		2	3	80.58	73.03	65.26	57.68	50.56
	2	4	77.86	66.42	54.50	43.41	33.82		2	4	79.59	72.12	64.53	57.13	50.16
	2	5	78.39	67.22	55.31	44.07	34.30		2	5	79.86	72.18	64.44	56.96	49.97
	2	6	76.51	65.44	53.90	43.08	33.66		2	6	78.83	70.70	62.71	55.18	48.27
grep	2	2	80.52	63.68	48.66	36.74	27.71	siena	2	2	71.29	47.54	30.17	19.05	12.20
	2	3	80.07	63.78	49.02	37.16	28.06		2	3	71.05	47.68	30.41	19.23	12.29
	2	4	79.14	63.05	48.58	36.92	27.94		2	4	70.57	47.68	30.51	19.30	12.33
	2	5	79.92	63.49	48.83	37.07	28.04		2	5	69.47	47.14	30.31	19.24	12.31
	2	6	78.87	62.77	48.38	36.81	27.90		2	6	68.58	46.51	30.01	19.13	12.29

GZIP and SIENA we just needed pairwise coverage; for GREP, three-way coverage; for NANOXML, three-way as well (obtained with ACTS). For SED it was sufficient to generate a four-way covering array to detect the same faults as the full TSL suite. In all cases pairwise coverage achieves at least 75 percent fault coverage.

Furthermore, all prioritisation strategies achieved similar fault detection rates. In 85.29 percent (90 percent for Java subjects only) of cases in Table 16 and 77.14 percent (75 percent for Java subjects only) of cases in Table 17, covering arrays ordered by pairwise criterion achieve best fault detection rates in comparison to higher-strength prioritisation criteria. Overall, the five-way criterion is the best producing best results in 85.29 percent of cases for covering arrays generated using CASA and in 85.71 percent of cases for covering arrays generated using ACTS. However, all prioritisation strategies produce best fault detection rates in at least 77.14 percent of cases presented in Tables 16 and 17.

Partially answering *RQ5*, we have found no clear consistency between the different prioritisation strategies. This might be partially due to the small number of faults available (up to 16 involving multi-valued parameters). However, pairwise coverage scaled well in comparison to higher-strength coverage prioritisation criteria.

Since higher strength CAs contain a larger number of test cases, comparing fault detection rates against percentages of test suite executed is not fair for lower strength covering arrays. To address this issue, Tables 19, 20 and 21 present the fault detection rate information against actual numbers of test cases executed, allowing direct comparison over all CAs: the tables show the percentage of detected faults after multiples of 10 test case executions (CAs smaller than the given number of executions are marked with -).

In 58.71 percent (56.10 percent for Java subjects only) of cases, CAs generated by CASA and ordered by the pairwise criterion achieve best fault detection rates. For CAs generated by ACTS, the pairwise strategy is the best, achieving best fault detection rates in 68.05 percent (72.09 percent for Java subjects only) cases. The six-way prioritisation criterion is slightly better for CAs generated using CASA, producing best fault detection rates in 72.90 percent of cases. However, all prioritisation strategies produce best fault detection rates in at least 58.06 percent of cases.

Moreover, pairwise CAs are no worse at early fault detection than higher-strength CAs. In 43 percent of cases presented in Tables 19 and 20, at least one pairwise ordered CA produces the best fault detection rate among all covering arrays for a given subject.

These results provide a mixed response to the remainder of *RQ5*: there is no dominant prioritisation criterion with respect to fault detection rate after a specific number of test executions; lower strength CAs produce fault detection rates comparable to those of higher strengths.

This suggests the following recommendation for best practice in prioritised combinatorial interaction testing: *given sufficient time and resources for testing, higher strength CAs under constraints are feasible and detect more faults. However, with limited time, lower strength CAs still provide a reasonable fault detection rate.*

When comparing CASA and ACTS covering array generation tools, there is no clear winner. GAcit achieves similar results. Analysing data in Tables 16 and 17 shows that, in 121 cases, CAs generated by CASA produce higher fault detection rates than CAs generated by ACTS, while in 129 cases the opposite is true. Test suites generated by CASA are sometimes better in terms of fault detection than the ones generated by ACTS, as is in the case of pairwise and three-way test suites for FLEX, where ACTS-generated test suites cover 92 and 96 percent of six-way interactions, while CASA-generated test suites cover 94 and 98 percent six-way interactions, respectively. However, for example, ACTS's four-way test suites for FLEX detect more faults than CASA—100 percent versus 96 percent (see Tables 16 and 17). Furthermore, previously generated minimal test suites for FLEX, shown in our previous work [21], cover more faults. Hence, two question arise: how many times should CASA be run? and which test suites should be chosen to increase the probability of high fault detection rates?

5.4 Greedy Scalability on Unconstrained Problems

In this section we address *RQ2* for unconstrained CIT. Table 22 presents the results that compare the executions times for the greedy algorithm on constrained and unconstrained CIT. As we can see, the worst case execution time is for six-way interaction testing of GZIP, for which the execution time is about 8 minutes (486.390 seconds in Table 22). Though this is considerably slower than the timings for

TABLE 16
Percentage of Detected Faults for All Versions of Subjects Used (CASA)

Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%	Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%
	2	2	86.0	94.0	94.0	94.0		2	2	76.19	90.48	95.24	95.24
	2	3	84.0	90.0	94.0	94.0		2	3	80.95	90.48	95.24	95.24
flex	2	4	86.0	92.0	94.0	94.0	sed	2	4	76.19	85.71	95.24	95.24
	2	5	90.0	92.0	94.0	94.0		2	5	85.71	90.48	95.24	95.24
	2	6	86.0	94.0	94.0	94.0		2	6	85.71	95.24	95.24	95.24
	3	2	96.0	98.0	98.0	98.0		3	2	90.48	95.24	95.24	95.24
	3	3	90.0	96.0	98.0	98.0		3	3	90.48	90.48	95.24	95.24
flex	3	4	94.0	96.0	98.0	98.0	sed	3	4	90.48	95.24	95.24	95.24
	3	5	94.0	98.0	98.0	98.0		3	5	85.71	90.48	90.48	95.24
	3	6	92.0	98.0	98.0	98.0		3	6	90.48	95.24	95.24	95.24
	4	2	92.0	92.0	96.0	96.0		4	2	85.71	90.48	100.0	100.0
	4	3	90.0	92.0	92.0	96.0		4	3	85.71	85.71	90.48	100.0
flex	4	4	94.0	96.0	96.0	96.0	sed	4	4	95.24	100.0	100.0	100.0
	4	5	92.0	96.0	96.0	96.0		4	5	90.48	90.48	100.0	100.0
	4	6	92.0	92.0	96.0	96.0		4	6	90.48	95.24	100.0	100.0
	5	2	96.0	96.0	96.0	96.0		5	2	85.71	90.48	95.24	100.0
	5	3	94.0	96.0	96.0	96.0		5	3	90.48	90.48	90.48	100.0
flex	5	4	94.0	94.0	94.0	96.0	sed	5	4	90.48	100.0	100.0	100.0
	5	5	92.0	92.0	96.0	96.0		5	5	85.71	100.0	100.0	100.0
	5	6	92.0	92.0	94.0	96.0		5	6	85.71	95.24	100.0	100.0
	6	2	100.0	100.0	100.0	100.0		6	2	-	-	-	-
	6	3	100.0	100.0	100.0	100.0		6	3	-	-	-	-
flex	6	4	100.0	100.0	100.0	100.0	sed	6	4	-	-	-	-
	6	5	100.0	100.0	100.0	100.0		6	5	-	-	-	-
	6	6	100.0	100.0	100.0	100.0		6	6	-	-	-	-
	2	2	50.0	100.0	100.0	100.0		2	2	80.0	100.0	100.0	100.0
	2	3	100.0	100.0	100.0	100.0		2	3	80.0	100.0	100.0	100.0
make	2	4	100.0	100.0	100.0	100.0	gzip	2	4	100.0	100.0	100.0	100.0
	2	5	50.0	100.0	100.0	100.0		2	5	80.0	80.0	100.0	100.0
	2	6	100.0	100.0	100.0	100.0		2	6	100.0	100.0	100.0	100.0
	3	2	100.0	100.0	100.0	100.0		3	2	100.0	100.0	100.0	100.0
	3	3	100.0	100.0	100.0	100.0		3	3	100.0	100.0	100.0	100.0
make	3	4	100.0	100.0	100.0	100.0	gzip	3	4	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		3	5	100.0	100.0	100.0	100.0
	3	6	100.0	100.0	100.0	100.0		3	6	100.0	100.0	100.0	100.0
	4	2	100.0	100.0	100.0	100.0		4	2	100.0	100.0	100.0	100.0
	4	3	100.0	100.0	100.0	100.0		4	3	100.0	100.0	100.0	100.0
make	4	4	100.0	100.0	100.0	100.0	gzip	4	4	100.0	100.0	100.0	100.0
	4	5	100.0	100.0	100.0	100.0		4	5	100.0	100.0	100.0	100.0
	4	6	100.0	100.0	100.0	100.0		4	6	100.0	100.0	100.0	100.0
	5	2	100.0	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
make	5	4	100.0	100.0	100.0	100.0	gzip	5	4	100.0	100.0	100.0	100.0
	5	5	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0
	6	2	100.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0
	6	3	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0
make	6	4	100.0	100.0	100.0	100.0	gzip	6	4	100.0	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0		6	5	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0		6	6	100.0	100.0	100.0	100.0
	2	2	75.0	83.33	83.33	83.33		2	2	50.0	68.75	87.5	87.5
	2	3	75.0	83.33	83.33	83.33		2	3	25.0	50.0	68.75	87.5
grep	2	4	75.0	75.0	83.33	83.33	nanoxml	2	4	6.25	50.0	87.5	87.5
	2	5	83.33	83.33	83.33	83.33		2	5	31.25	50.0	87.5	87.5
	2	6	75.0	83.33	83.33	83.33		2	6	25.0	68.75	68.75	87.5
	3	2	91.67	100.0	100.0	100.0		3	2	93.75	93.75	93.75	93.75
	3	3	91.67	91.67	91.67	100.0		3	3	87.5	93.75	93.75	93.75
grep	3	4	83.33	91.67	91.67	100.0	nanoxml	3	4	68.75	93.75	93.75	93.75
	3	5	100.0	100.0	100.0	100.0		3	5	87.5	93.75	93.75	93.75
	3	6	100.0	100.0	100.0	100.0		3	6	93.75	93.75	93.75	93.75
	4	2	100.0	100.0	100.0	100.0		4	2	68.75	93.75	93.75	93.75
	4	3	91.67	100.0	100.0	100.0		4	3	87.5	93.75	93.75	93.75
grep	4	4	91.67	91.67	100.0	100.0	nanoxml	4	4	93.75	93.75	93.75	93.75
	4	5	100.0	100.0	100.0	100.0		4	5	93.75	93.75	93.75	93.75
	4	6	91.67	91.67	91.67	100.0		4	6	93.75	93.75	93.75	93.75
	5	2	91.67	100.0	100.0	100.0		5	2	87.5	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
grep	5	4	91.67	91.67	100.0	100.0	nanoxml	5	4	93.75	93.75	93.75	100.0
	5	5	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
	5	6	91.67	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0
	6	2	91.67	100.0	100.0	100.0		6	2	93.75	100.0	100.0	100.0
	6	3	91.67	91.67	100.0	100.0		6	3	100.0	100.0	100.0	100.0
grep	6	4	91.67	91.67	100.0	100.0	nanoxml	6	4	93.75	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0		6	5	93.75	100.0	100.0	100.0
	6	6	91.67	100.0	100.0	100.0		6	6	93.75	100.0	100.0	100.0
	2	2	100.0	100.0	100.0	100.0	siena	4	5	100.0	100.0	100.0	100.0
	2	3	50.0	75.0	100.0	100.0		4	6	100.0	100.0	100.0	100.0
siena	2	4	75.0	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0
	2	5	75.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
	2	6	100.0	100.0	100.0	100.0	siena	5	4	100.0	100.0	100.0	100.0
	3	2	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
	3	3	100.0	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0
siena	3	4	75.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0
	3	6	100.0	100.0	100.0	100.0		6	4	100.0	100.0	100.0	100.0
	4	2	75.0	100.0	100.0	100.0	siena	6	5	100.0	100.0	100.0	100.0
siena	4	3	100.0	100.0	100.0	100.0		6	6	100.0	100.0	100.0	100.0
	4	4	100.0	100.0	100.0	100.0							

TABLE 17
Percentage of Detected Faults for All Versions of Subjects Used (ACTS)

Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%	Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%
	2	2	90.0	92.0	92.0	92.0		2	2	76.19	85.71	90.48	90.48
	2	3	90.0	90.0	92.0	92.0		2	3	85.71	90.48	90.48	90.48
flex	2	4	88.0	92.0	92.0	92.0	sed	2	4	85.71	90.48	90.48	90.48
	2	5	90.0	92.0	92.0	92.0		2	5	76.19	90.48	90.48	90.48
	2	6	72.0	92.0	92.0	92.0		2	6	80.95	85.71	90.48	90.48
	3	2	90.0	94.0	94.0	96.0		3	2	85.71	90.48	90.48	95.24
	3	3	88.0	94.0	94.0	96.0		3	3	90.48	95.24	95.24	95.24
flex	3	4	92.0	92.0	92.0	96.0	sed	3	4	90.48	95.24	95.24	95.24
	3	5	92.0	94.0	94.0	96.0		3	5	90.48	95.24	95.24	95.24
	3	6	90.0	94.0	94.0	96.0		3	6	90.48	90.48	95.24	95.24
	4	2	92.0	100.0	100.0	100.0		4	2	90.48	95.24	100.0	100.0
	4	3	94.0	98.0	98.0	100.0		4	3	95.24	100.0	100.0	100.0
flex	4	4	96.0	96.0	100.0	100.0	sed	4	4	90.48	100.0	100.0	100.0
	4	5	94.0	100.0	100.0	100.0		4	5	100.0	100.0	100.0	100.0
	4	6	94.0	96.0	100.0	100.0		4	6	100.0	100.0	100.0	100.0
	5	2	94.0	96.0	96.0	100.0		5	2	100.0	100.0	100.0	100.0
	5	3	94.0	96.0	96.0	100.0		5	3	95.24	95.24	95.24	100.0
flex	5	4	94.0	96.0	100.0	100.0	sed	5	4	95.24	95.24	95.24	100.0
	5	5	94.0	100.0	100.0	100.0		5	5	90.48	90.48	95.24	100.0
	5	6	100.0	100.0	100.0	100.0		5	6	85.71	95.24	95.24	100.0
	6	2	94.0	96.0	96.0	100.0		6	2	95.24	95.24	95.24	100.0
	6	3	96.0	96.0	96.0	100.0		6	3	100.0	100.0	100.0	100.0
flex	6	4	94.0	96.0	96.0	100.0	sed	6	4	90.48	95.24	95.24	100.0
	6	5	100.0	100.0	100.0	100.0		6	5	85.71	100.0	100.0	100.0
	6	6	96.0	96.0	96.0	100.0		6	6	90.48	90.48	95.24	100.0
	2	2	100.0	100.0	100.0	100.0		2	2	100.0	100.0	100.0	100.0
	2	3	100.0	100.0	100.0	100.0		2	3	100.0	100.0	100.0	100.0
make	2	4	100.0	100.0	100.0	100.0	gzip	2	4	80.0	100.0	100.0	100.0
	2	5	100.0	100.0	100.0	100.0		2	5	80.0	100.0	100.0	100.0
	2	6	100.0	100.0	100.0	100.0		2	6	100.0	100.0	100.0	100.0
	3	2	100.0	100.0	100.0	100.0		3	2	100.0	100.0	100.0	100.0
	3	3	100.0	100.0	100.0	100.0		3	3	100.0	100.0	100.0	100.0
make	3	4	100.0	100.0	100.0	100.0	gzip	3	4	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		3	5	100.0	100.0	100.0	100.0
	3	6	100.0	100.0	100.0	100.0		3	6	100.0	100.0	100.0	100.0
	4	2	100.0	100.0	100.0	100.0		4	2	100.0	100.0	100.0	100.0
	4	3	100.0	100.0	100.0	100.0		4	3	100.0	100.0	100.0	100.0
make	4	4	100.0	100.0	100.0	100.0	gzip	4	4	100.0	100.0	100.0	100.0
	4	5	100.0	100.0	100.0	100.0		4	5	100.0	100.0	100.0	100.0
	4	6	100.0	100.0	100.0	100.0		4	6	100.0	100.0	100.0	100.0
	5	2	100.0	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
make	5	4	100.0	100.0	100.0	100.0	gzip	5	4	100.0	100.0	100.0	100.0
	5	5	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0
	6	2	100.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0
	6	3	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0
make	6	4	100.0	100.0	100.0	100.0	gzip	6	4	100.0	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0		6	5	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0		6	6	100.0	100.0	100.0	100.0
	2	2	75.0	83.33	83.33	91.67		2	2	68.75	68.75	87.5	87.5
	2	3	91.67	91.67	91.67	91.67		2	3	6.25	43.75	68.75	87.5
grep	2	4	91.67	91.67	91.67	91.67	nanoxml	2	4	25.0	43.75	68.75	87.5
	2	5	75.0	91.67	91.67	91.67		2	5	68.75	87.5	87.5	87.5
	2	6	91.67	91.67	91.67	91.67		2	6	50.0	68.75	87.5	87.5
	3	2	91.67	91.67	91.67	100.0		3	2	93.75	100.0	100.0	100.0
	3	3	91.67	91.67	91.67	100.0		3	3	100.0	100.0	100.0	100.0
grep	3	4	83.33	100.0	100.0	100.0	nanoxml	3	4	68.75	93.75	100.0	100.0
	3	5	83.33	91.67	91.67	100.0		3	5	93.75	93.75	100.0	100.0
	3	6	83.33	91.67	91.67	100.0		3	6	87.5	93.75	100.0	100.0
	4	2	100.0	100.0	100.0	100.0		4	2	93.75	93.75	93.75	100.0
	4	3	91.67	91.67	91.67	100.0		4	3	75.0	100.0	100.0	100.0
grep	4	4	100.0	100.0	100.0	100.0	nanoxml	4	4	68.75	87.5	93.75	100.0
	4	5	100.0	100.0	100.0	100.0		4	5	68.75	87.5	93.75	100.0
	4	6	100.0	100.0	100.0	100.0		4	6	93.75	93.75	93.75	100.0
	5	2	100.0	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0		5	3	93.75	93.75	100.0	100.0
grep	5	4	100.0	100.0	100.0	100.0	nanoxml	5	4	93.75	100.0	100.0	100.0
	5	5	100.0	100.0	100.0	100.0		5	5	93.75	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0		5	6	93.75	93.75	100.0	100.0
	6	2	100.0	100.0	100.0	100.0		6	2	93.75	93.75	93.75	100.0
	6	3	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0
grep	6	4	100.0	100.0	100.0	100.0	nanoxml	6	4	93.75	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0		6	5	93.75	93.75	100.0	100.0
	6	6	100.0	100.0	100.0	100.0		6	6	93.75	93.75	93.75	100.0
	2	2	50.0	50.0	75.0	75.0	siena	4	5	100.0	100.0	100.0	100.0
	2	3	50.0	50.0	75.0	75.0		4	6	100.0	100.0	100.0	100.0
siena	2	4	50.0	50.0	75.0	75.0		5	2	100.0	100.0	100.0	100.0
	2	5	75.0	75.0	75.0	75.0		5	3	100.0	100.0	100.0	100.0
	2	6	50.0	75.0	75.0	75.0	siena	5	4	100.0	100.0	100.0	100.0
	3	2	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
	3	3	100.0	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0
siena	3	4	100.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0
	3	6	75.0	100.0	100.0	100.0	siena	6	4	100.0	100.0	100.0	100.0
	4	2	100.0	100.0	100.0	100.0		6	5	100.0	100.0	100.0	100.0
siena	4	3	100.0	100.0	100.0	100.0		6	6	100.0	100.0	100.0	100.0
	4	4	100.0	100.0	100.0	100.0							

TABLE 18
Percentage of Detected Faults for All Versions of Subjects Used (GAcit)

Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%	Subjects	Gen. Crit.	Prio. Crit.	% of Test Suite Executed 25%	50%	75%	100%
flex	2	2	88.0	92.0	92.0	92.0	gzip	2	2	80.0	100.0	100.0	100.0
	2	3	88.0	92.0	92.0	92.0		2	3	100.0	100.0	100.0	100.0
	2	4	92.0	92.0	92.0	92.0		2	4	100.0	100.0	100.0	100.0
	2	5	90.0	90.0	90.0	92.0		2	5	80.0	100.0	100.0	100.0
	2	6	88.0	92.0	92.0	92.0		2	6	80.0	100.0	100.0	100.0
make	2	2	100.0	100.0	100.0	100.0	nanoxml	2	2	25.0	68.75	87.5	87.5
	2	3	50.0	100.0	100.0	100.0		2	3	25.0	68.75	68.75	87.5
	2	4	100.0	100.0	100.0	100.0		2	4	50.0	50.0	68.75	87.5
	2	5	100.0	100.0	100.0	100.0		2	5	68.75	68.75	87.5	87.5
	2	6	100.0	100.0	100.0	100.0		2	6	25.0	87.5	87.5	87.5
grep	2	2	75.0	83.33	91.67	91.67	siena	2	2	75.0	100.0	100.0	100.0
	2	3	50.0	83.33	91.67	91.67		2	3	75.0	100.0	100.0	100.0
	2	4	91.67	91.67	91.67	91.67		2	4	50.0	75.0	75.0	100.0
	2	5	83.33	91.67	91.67	91.67		2	5	50.0	50.0	75.0	100.0
	2	6	75.0	83.33	91.67	91.67		2	6	75.0	75.0	75.0	100.0

constrained problems (all of which terminated in under one second), it is by no means infeasible. Moreover, runtimes for unconstrained and constrained CIT models for SIENA, in contrast to other subjects, are comparable. We note that size reduction (in terms of the number of parameters) due to constraints for this subject is the lowest, as shown in Table 7.

The 'forbidden tuples' constraint handling option turned out to be efficient, suggesting that majority of time spent by ACTS on constrained CIT problems was devoted to constraint satisfaction. Therefore, it is not surprising that ACTS remains as reasonably scalable on unconstrained problems as it is on constrained problems (by suitable deployment of the 'forbidden tuples' constraint handling procedure).

Table 22 also reveals the considerable influence of the constraints on the overall CIT test process. While the greedy algorithm is feasible for unconstrained problems, it produces between one and three orders of magnitude more test cases for unconstrained CIT problems. This further underscores the importance of constraints, suggesting that future work on CIT should explore ways of increasing the amount of constraint information available to the CIT algorithm. One way to do this would be through the exploration of soft constraints, which have hitherto been less widely studied in the CIT literature.

5.5 Greedy vs SA vs GA

Overall, we observed little difference between test suites generated by CASA and ACTS in terms of efficiency, t-way coverage and fault detection rates. GAcit was significantly worse than the other two approaches. Greedily generated test suites are usually bigger (and generated in a shorter amount of time) than those generated by both CASA and GAcit. Therefore, they cover more t-way interactions. However, they do not always discover faults as quickly as CASA does when the test suite is prioritised.

6 Future Work

Since SA and Greedy algorithms produce good, small test suites, a question arises: which test suites are the best in terms of coverage and fault detection rates? Moreover, is there a way of making an SA-based CIT tool produce deterministic results? Determinism is important in the industry. For example, one might want to extend an existing test suite to cover all t-way interactions when a new component is added to the system. Seeding can be used and a test suite built around the existing tests, or incremental approaches may be in order [37].

Moreover, further investigation has shown that the efficiency of the tools used highly relies on constraint handling. Constraint handling in ACTS is much faster on our subjects when the 'forbidden tuples' constraint handling technique is used (that is a list of interactions that cannot occur together is generated). The default method, that is, using a CSP solver (a constraint solver for Constraint Satisfaction Problems, specialising in various types of constraints) provides substantial overhead. For GREP result for six-way was not generated within several hours, but it was generated within seconds when 'forbidden tuples' method was used. It is unclear if there are certain parameter sizes of software systems where this result will differ. Subjects with high number of constraints also significantly decreased efficiency of the Greedy approach.

7 Threats to Validity

We tested the scalability of CIT techniques for test case generation in the presence of constraints. To test two of the most popular techniques, one based on simulated-annealing and another using a greedy approach, we used two state-of-the-art CIT tools. Our results might thus depend on the implementation of the algorithms within these two tools, however given that these are considered state-of-the-art we believe that they are representative of their respective algorithms. As for the GA, since they have not been used as much in the literature, there is a risk that our results for GAcit have more dependence on the specific implementation.

As our results indicated, the choice of constraint handling method adopted has a dramatic impact on the performance of the greedy approach. Thus direct comparison of SA and greedy in presence of constraints is clearly dependent on the constraint solving method used.

The CASA approach is stochastic in nature, since it uses simulated annealing. Therefore, its results can vary from one execution to the next. Same is true for GAcit. Since Greedy is deterministic, there is no value to be gained from an inferential statistical analysis (the Greedy results are not drawn from a population). However, in order to cater for

TABLE 19
Percentage of Detected Faults up to Multiples of 10 Test Case Executions (CASA)

Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60
flex	2	2	90.0	94.0	-	-	-	-
	2	3	90.0	94.0	-	-	-	-
	2	4	88.0	94.0	-	-	-	-
	2	5	92.0	94.0	-	-	-	-
	2	6	88.0	94.0	-	-	-	-
flex	3	2	90.0	98.0	98.0	98.0	98.0	-
	3	3	90.0	96.0	96.0	98.0	98.0	-
	3	4	94.0	96.0	96.0	98.0	98.0	-
	3	5	94.0	94.0	98.0	98.0	98.0	-
	3	6	90.0	92.0	98.0	98.0	98.0	-
flex	4	2	92.0	92.0	92.0	92.0	92.0	92.0
	4	3	90.0	90.0	90.0	92.0	92.0	92.0
	4	4	90.0	94.0	94.0	94.0	96.0	96.0
	4	5	92.0	92.0	92.0	92.0	92.0	96.0
	4	6	90.0	92.0	92.0	92.0	92.0	96.0
flex	5	2	90.0	96.0	96.0	96.0	96.0	96.0
	5	3	90.0	92.0	94.0	94.0	94.0	94.0
	5	4	92.0	94.0	94.0	94.0	94.0	94.0
	5	5	88.0	90.0	92.0	92.0	92.0	92.0
	5	6	88.0	88.0	90.0	90.0	92.0	92.0
flex	6	2	90.0	98.0	100.0	100.0	100.0	100.0
	6	3	88.0	92.0	96.0	96.0	96.0	100.0
	6	4	90.0	92.0	92.0	94.0	94.0	96.0
	6	5	90.0	90.0	96.0	96.0	98.0	100.0
	6	6	92.0	92.0	94.0	98.0	98.0	100.0
make	2	2	-	-	-	-	-	-
	2	3	-	-	-	-	-	-
	2	4	-	-	-	-	-	-
	2	5	-	-	-	-	-	-
	2	6	-	-	-	-	-	-
make	3	2	100.0	-	-	-	-	-
	3	3	100.0	-	-	-	-	-
	3	4	100.0	-	-	-	-	-
	3	5	100.0	-	-	-	-	-
	3	6	100.0	-	-	-	-	-
make	4	2	100.0	100.0	-	-	-	-
	4	3	100.0	100.0	-	-	-	-
	4	4	100.0	100.0	-	-	-	-
	4	5	100.0	100.0	-	-	-	-
	4	6	100.0	100.0	-	-	-	-
make	5	2	100.0	100.0	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0	100.0	100.0
	5	4	100.0	100.0	100.0	100.0	100.0	100.0
	5	5	100.0	100.0	100.0	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0	100.0	100.0
make	6	2	100.0	100.0	100.0	100.0	100.0	100.0
	6	3	100.0	100.0	100.0	100.0	100.0	100.0
	6	4	100.0	100.0	100.0	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0	100.0	100.0
grep	2	2	75.0	83.33	83.33	83.33	-	-
	2	3	75.0	83.33	83.33	83.33	-	-
	2	4	75.0	75.0	83.33	83.33	-	-
	2	5	83.33	83.33	83.33	83.33	-	-
	2	6	75.0	83.33	83.33	83.33	-	-
grep	3	2	75.0	83.33	91.67	91.67	91.67	91.67
	3	3	66.67	83.33	83.33	91.67	91.67	91.67
	3	4	75.0	83.33	83.33	83.33	91.67	91.67
	3	5	75.0	83.33	83.33	100.0	100.0	100.0
	3	6	83.33	83.33	100.0	100.0	100.0	100.0
grep	4	2	83.33	83.33	83.33	83.33	83.33	91.67
	4	3	66.67	66.67	75.0	91.67	91.67	91.67
	4	4	75.0	83.33	91.67	91.67	91.67	91.67
	4	5	83.33	83.33	83.33	83.33	83.33	91.67
	4	6	83.33	83.33	91.67	91.67	91.67	91.67
grep	5	2	83.33	83.33	91.67	91.67	91.67	91.67
	5	3	91.67	100.0	100.0	100.0	100.0	100.0
	5	4	83.33	83.33	91.67	91.67	91.67	91.67
	5	5	75.0	83.33	83.33	91.67	91.67	91.67
	5	6	75.0	83.33	83.33	91.67	91.67	91.67
grep	6	2	75.0	83.33	83.33	83.33	83.33	83.33
	6	3	75.0	83.33	83.33	83.33	83.33	83.33
	6	4	75.0	75.0	75.0	83.33	91.67	91.67
	6	5	91.67	91.67	100.0	100.0	100.0	100.0
	6	6	75.0	75.0	75.0	75.0	75.0	83.33
siena	2	2	75.0	-	-	-	-	-
	2	3	100.0	-	-	-	-	-
	2	4	100.0	-	-	-	-	-
	2	5	100.0	-	-	-	-	-
	2	6	100.0	-	-	-	-	-
siena	3	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	3	75.0	100.0	100.0	100.0	100.0	100.0
	3	4	75.0	100.0	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0	100.0	100.0
	3	6	75.0	100.0	100.0	100.0	100.0	100.0
siena	4	2	75.0	75.0	75.0	75.0	75.0	100.0
	4	3	100.0	100.0	100.0	100.0	100.0	100.0
	4	4	50.0	75.0	75.0	100.0	100.0	100.0

Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60
sed	2	2	66.67	85.71	90.48	95.24	95.24	-
	2	3	71.43	85.71	90.48	95.24	95.24	-
	2	4	71.43	85.71	85.71	95.24	95.24	-
	2	5	66.67	85.71	90.48	95.24	95.24	-
	2	6	76.19	95.24	95.24	95.24	95.24	-
sed	3	2	71.43	85.71	85.71	90.48	90.48	95.24
	3	3	66.67	80.95	85.71	90.48	90.48	90.48
	3	4	71.43	85.71	90.48	90.48	90.48	90.48
	3	5	71.43	76.19	76.19	85.71	90.48	90.48
	3	6	57.14	66.67	85.71	90.48	95.24	95.24
sed	4	2	71.43	71.43	80.95	80.95	80.95	85.71
	4	3	71.43	80.95	85.71	85.71	85.71	85.71
	4	4	66.67	76.19	80.95	85.71	85.71	85.71
	4	5	80.95	80.95	85.71	85.71	85.71	85.71
	4	6	80.95	90.48	90.48	90.48	90.48	90.48
sed	5	2	76.19	76.19	80.95	80.95	80.95	80.95
	5	3	57.14	76.19	76.19	80.95	80.95	80.95
	5	4	66.67	76.19	85.71	85.71	85.71	85.71
	5	5	57.14	66.67	76.19	80.95	80.95	80.95
	5	6	71.43	76.19	76.19	76.19	85.71	85.71
sed	6	2	-	-	-	-	-	-
	6	3	-	-	-	-	-	-
	6	4	-	-	-	-	-	-
	6	5	-	-	-	-	-	-
	6	6	-	-	-	-	-	-
gzip	2	2	100.0	-	-	-	-	-
	2	3	100.0	-	-	-	-	-
	2	4	100.0	-	-	-	-	-
	2	5	80.0	-	-	-	-	-
	2	6	100.0	-	-	-	-	-
gzip	3	2	100.0	100.0	100.0	100.0	-	-
	3	3	100.0	100.0	100.0	100.0	-	-
	3	4	100.0	100.0	100.0	100.0	-	-
	3	5	100.0	100.0	100.0	100.0	-	-
	3	6	100.0	100.0	100.0	100.0	-	-
gzip	4	2	100.0	100.0	100.0	100.0	100.0	100.0
	4	3	100.0	100.0	100.0	100.0	100.0	100.0
	4	4	100.0	100.0	100.0	100.0	100.0	100.0
	4	5	100.0	100.0	100.0	100.0	100.0	100.0
	4	6	80.0	100.0	100.0	100.0	100.0	100.0
gzip	5	2	100.0	100.0	100.0	100.0	100.0	100.0
	5	3	80.0	80.0	100.0	100.0	100.0	100.0
	5	4	100.0	100.0	100.0	100.0	100.0	100.0
	5	5	80.0	100.0	100.0	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0	100.0	100.0
gzip	6	2	80.0	100.0	100.0	100.0	100.0	100.0
	6	3	80.0	100.0	100.0	100.0	100.0	100.0
	6	4	100.0	100.0	100.0	100.0	100.0	100.0
	6	5	80.0	80.0	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0	100.0	100.0
nanoxml	2	2	-	-	-	-	-	-
	2	3	-	-	-	-	-	-
	2	4	-	-	-	-	-	-
	2	5	-	-	-	-	-	-
	2	6	-	-	-	-	-	-
nanoxml	3	2	93.75	-	-	-	-	-
	3	3	93.75	-	-	-	-	-
	3	4	93.75	-	-	-	-	-
	3	5	93.75	-	-	-	-	-
	3	6	93.75	-	-	-	-	-
nanoxml	4	2	68.75	93.75	93.75	-	-	-
	4	3	87.5	93.75	93.75	-	-	-
	4	4	93.75	93.75	93.75	-	-	-
	4	5	93.75	93.75	93.75	-	-	-
	4	6	93.75	93.75	93.75	-	-	-
nanoxml	5	2	87.5	93.75	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0	100.0	100.0
	5	4	87.5	93.75	93.75	93.75	93.75	100.0
	5	5	93.75	100.0	100.0	100.0	100.0	100.0
	5	6	93.75	100.0	100.0	100.0	100.0	100.0
nanoxml	6	2	75.0	93.75	100.0	100.0	100.0	100.0
	6	3	87.5	100.0	100.0	100.0	100.0	100.0
	6	4	75.0	93.75	100.0	100.0	100.0	100.0
	6	5	87.5	93.75	100.0	100.0	100.0	100.0
	6	6	93.75	93.75	100.0	100.0	100.0	100.0
siena	4	5	50.0	75.0	100.0	100.0	100.0	100.0
	4	6	100.0	100.0	100.0	100.0	100.0	100.0
siena	5	2	75.0	75.0	75.0	75.0	75.0	75.0
	5	3	75.0	100.0	100.0	100.0	100.0	100.0
	5	4	100.0	100.0	100.0	100.0	100.0	100.0
	5	5	75.0	100.0	100.0	100.0	100.0	100.0
	5	6	75.0	100.0	100.0	100.0	100.0	100.0
siena	6	2	75.0	100.0	100.0	100.0	100.0	100.0
	6	3	75.0	75.0	75.0	100.0	100.0	100.0
	6	4	75.0	75.0	100.0	100.0	100.0	100.0
	6	5	50.0	100.0	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0	100.0	100.0

TABLE 20
Percentage of Detected Faults up to Multiples of 10 Test Case Executions (ACTS)

Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60	Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60
	2	2	90.0	92.0	-	-	-	-		2	2	71.43	80.95	85.71	90.48	90.48	-
	2	3	90.0	92.0	-	-	-	-		2	3	80.95	85.71	90.48	90.48	90.48	-
flex	2	4	88.0	92.0	-	-	-	-	sed	2	4	76.19	85.71	90.48	90.48	90.48	-
	2	5	90.0	92.0	-	-	-	-		2	5	71.43	85.71	90.48	90.48	90.48	-
	2	6	90.0	92.0	-	-	-	-		2	6	71.43	85.71	85.71	90.48	90.48	-
	3	2	90.0	92.0	94.0	94.0	94.0	94.0		3	2	80.95	80.95	85.71	85.71	85.71	90.48
	3	3	86.0	90.0	94.0	94.0	94.0	94.0		3	3	76.19	85.71	90.48	90.48	95.24	95.24
flex	3	4	90.0	92.0	92.0	92.0	92.0	96.0	sed	3	4	80.95	85.71	90.48	90.48	90.48	95.24
	3	5	86.0	92.0	94.0	94.0	94.0	96.0		3	5	80.95	80.95	90.48	90.48	95.24	95.24
	3	6	86.0	94.0	94.0	94.0	94.0	96.0		3	6	71.43	85.71	90.48	90.48	90.48	90.48
	4	2	88.0	92.0	92.0	96.0	100.0	100.0		4	2	57.14	80.95	90.48	90.48	90.48	90.48
	4	3	92.0	94.0	94.0	94.0	98.0	98.0		4	3	52.38	80.95	80.95	85.71	90.48	90.48
flex	4	4	94.0	96.0	96.0	96.0	96.0	96.0	sed	4	4	66.67	76.19	76.19	85.71	90.48	90.48
	4	5	90.0	90.0	94.0	100.0	100.0	100.0		4	5	42.86	71.43	85.71	95.24	95.24	95.24
	4	6	90.0	92.0	92.0	96.0	96.0	96.0		4	6	71.43	76.19	80.95	90.48	90.48	95.24
	5	2	88.0	90.0	92.0	94.0	94.0	94.0		5	2	66.67	80.95	90.48	95.24	100.0	100.0
	5	3	86.0	90.0	90.0	92.0	92.0	94.0		5	3	76.19	85.71	85.71	95.24	95.24	95.24
flex	5	4	90.0	92.0	94.0	94.0	94.0	94.0	sed	5	4	61.9	71.43	80.95	80.95	90.48	95.24
	5	5	78.0	94.0	94.0	94.0	94.0	94.0		5	5	66.67	80.95	80.95	80.95	80.95	80.95
	5	6	88.0	92.0	94.0	94.0	98.0	100.0		5	6	52.38	76.19	80.95	80.95	80.95	85.71
	6	2	90.0	90.0	90.0	90.0	94.0	94.0		6	2	76.19	85.71	90.48	95.24	95.24	95.24
	6	3	90.0	90.0	92.0	94.0	96.0	96.0		6	3	66.67	80.95	80.95	95.24	95.24	95.24
flex	6	4	90.0	94.0	94.0	94.0	94.0	94.0	sed	6	4	71.43	71.43	80.95	85.71	85.71	85.71
	6	5	86.0	98.0	98.0	98.0	98.0	98.0		6	5	61.9	80.95	80.95	80.95	85.71	85.71
	6	6	90.0	92.0	94.0	96.0	96.0	96.0		6	6	61.9	80.95	80.95	80.95	80.95	85.71
	2	2	-	-	-	-	-	-		2	2	100.0	100.0	-	-	-	-
	2	3	-	-	-	-	-	-		2	3	100.0	100.0	-	-	-	-
make	2	4	-	-	-	-	-	-	gzip	2	4	100.0	100.0	-	-	-	-
	2	5	-	-	-	-	-	-		2	5	80.0	100.0	-	-	-	-
	2	6	-	-	-	-	-	-		2	6	100.0	100.0	-	-	-	-
	3	2	100.0	-	-	-	-	-		3	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	3	100.0	-	-	-	-	-		3	3	100.0	100.0	100.0	100.0	100.0	100.0
make	3	4	100.0	-	-	-	-	-	gzip	3	4	100.0	100.0	100.0	100.0	100.0	100.0
	3	5	100.0	-	-	-	-	-		3	5	100.0	100.0	100.0	100.0	100.0	100.0
	3	6	100.0	-	-	-	-	-		3	6	100.0	100.0	100.0	100.0	100.0	100.0
	4	2	100.0	100.0	100.0	100.0	-	-		4	2	100.0	100.0	100.0	100.0	100.0	100.0
	4	3	100.0	100.0	100.0	100.0	-	-		4	3	100.0	100.0	100.0	100.0	100.0	100.0
make	4	4	100.0	100.0	100.0	100.0	-	-	gzip	4	4	100.0	100.0	100.0	100.0	100.0	100.0
	4	5	100.0	100.0	100.0	100.0	-	-		4	5	80.0	80.0	100.0	100.0	100.0	100.0
	4	6	100.0	100.0	100.0	100.0	-	-		4	6	80.0	100.0	100.0	100.0	100.0	100.0
	5	2	100.0	100.0	100.0	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0	100.0	100.0
	5	3	100.0	100.0	100.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0	100.0	100.0
make	5	4	100.0	100.0	100.0	100.0	100.0	100.0	gzip	5	4	100.0	100.0	100.0	100.0	100.0	100.0
	5	5	100.0	100.0	100.0	100.0	100.0	100.0		5	5	80.0	100.0	100.0	100.0	100.0	100.0
	5	6	100.0	100.0	100.0	100.0	100.0	100.0		5	6	80.0	80.0	100.0	100.0	100.0	100.0
	6	2	100.0	100.0	100.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0	100.0	100.0
	6	3	100.0	100.0	100.0	100.0	100.0	100.0		6	3	80.0	100.0	100.0	100.0	100.0	100.0
make	6	4	100.0	100.0	100.0	100.0	100.0	100.0	gzip	6	4	100.0	100.0	100.0	100.0	100.0	100.0
	6	5	100.0	100.0	100.0	100.0	100.0	100.0		6	5	100.0	100.0	100.0	100.0	100.0	100.0
	6	6	100.0	100.0	100.0	100.0	100.0	100.0		6	6	80.0	100.0	100.0	100.0	100.0	100.0
	2	2	50.0	83.33	83.33	91.67	-	-		2	2	-	-	-	-	-	-
	2	3	91.67	91.67	91.67	91.67	-	-		2	3	-	-	-	-	-	-
grep	2	4	91.67	91.67	91.67	91.67	-	-	nanoxml	2	4	-	-	-	-	-	-
	2	5	75.0	91.67	91.67	91.67	-	-		2	5	-	-	-	-	-	-
	2	6	91.67	91.67	91.67	91.67	-	-		2	6	-	-	-	-	-	-
	3	2	83.33	91.67	91.67	91.67	91.67	91.67		3	2	93.75	100.0	-	-	-	-
	3	3	66.67	83.33	83.33	91.67	91.67	91.67		3	3	100.0	100.0	-	-	-	-
grep	3	4	75.0	83.33	83.33	83.33	83.33	91.67	nanoxml	3	4	87.5	100.0	-	-	-	-
	3	5	83.33	83.33	83.33	83.33	91.67	91.67		3	5	93.75	100.0	-	-	-	-
	3	6	75.0	75.0	75.0	83.33	91.67	91.67		3	6	93.75	100.0	-	-	-	-
	4	2	83.33	91.67	91.67	91.67	91.67	100.0		4	2	93.75	93.75	100.0	-	-	-
	4	3	83.33	91.67	91.67	91.67	91.67	91.67		4	3	75.0	100.0	100.0	-	-	-
grep	4	4	75.0	83.33	91.67	91.67	91.67	91.67	nanoxml	4	4	87.5	87.5	93.75	-	-	-
	4	5	75.0	83.33	83.33	83.33	100.0	100.0		4	5	68.75	93.75	93.75	-	-	-
	4	6	91.67	100.0	100.0	100.0	100.0	100.0		4	6	93.75	93.75	100.0	-	-	-
	5	2	75.0	83.33	91.67	100.0	100.0	100.0		5	2	100.0	100.0	100.0	100.0	100.0	100.0
	5	3	75.0	75.0	75.0	75.0	75.0	91.67		5	3	93.75	93.75	93.75	93.75	100.0	100.0
grep	5	4	75.0	75.0	83.33	91.67	91.67	91.67	nanoxml	5	4	93.75	93.75	100.0	100.0	100.0	100.0
	5	5	83.33	91.67	91.67	91.67	91.67	100.0		5	5	93.75	93.75	100.0	100.0	100.0	100.0
	5	6	91.67	91.67	100.0	100.0	100.0	100.0		5	6	87.5	93.75	93.75	93.75	100.0	100.0
	6	2	66.67	83.33	100.0	100.0	100.0	100.0		6	2	93.75	93.75	93.75	93.75	93.75	93.75
	6	3	75.0	91.67	91.67	91.67	91.67	91.67		6	3	100.0	100.0	100.0	100.0	100.0	100.0
grep	6	4	83.33	83.33	91.67	91.67	100.0	100.0	nanoxml	6	4	93.75	93.75	93.75	100.0	100.0	100.0
	6	5	83.33	83.33	83.33	83.33	83.33	83.33		6	5	87.5	93.75	93.75	100.0	100.0	100.0
	6	6	75.0	83.33	83.33	83.33	91.67	91.67		6	6	93.75	93.75	93.75	93.75	93.75	100.0
	2	2	50.0	75.0	-	-	-	-	siena	4	5	100.0	100.0	100.0	100.0	100.0	100.0
	2	3	50.0	75.0	-	-	-	-		4	6	75.0	100.0	100.0	100.0	100.0	100.0
siena	2	4	50.0	75.0	-	-	-	-		5	2	75.0	100.0	100.0	100.0	100.0	100.0
	2	5	75.0	75.0	-	-	-	-		5	3	100.0	100.0	100.0	100.0	100.0	100.0
	2	6	75.0	75.0	-	-	-	-	siena	5	4	100.0	100.0	100.0	100.0	100.0	100.0
	3	2	75.0	100.0	100.0	100.0	100.0	100.0		5	5	75.0	100.0	100.0	100.0	100.0	100.0
	3	3	75.0	100.0	100.0	100.0	100.0	100.0		5	6	100.0	100.0	100.0	100.0	100.0	100.0
siena	3	4	100.0	100.0	100.0	100.0	100.0	100.0		6	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	5	50.0	100.0	100.0	100.0	100.0	100.0		6	3	100.0	100.0	100.0	100.0	100.0	100.0
	3	6	50.0	75.0	100.0	100.0	100.0	100.0	siena	6	4	100.0	100.0	100.0	100.0	100.0	100.0
	4	2	100.0	100.0	100.0	100.0	100.0	100.0		6	5	50.0	75.0	75.0	75.0	75.0	75.0
siena	4	3	75.0	75.0	75.0	100.0	100.0	100.0		6	6	75.0	75.0	75.0	100.0	100.0	100.0
	4	4	50.0	75.0	100.0	100.0	100.0	100.0									

TABLE 21
Percentage of Detected Faults up to Multiples of 10 Test Case Executions (GAcit)

Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60	Subjects	Gen. Crit.	Prio. Crit.	Num. of Test Cases Executed 10	20	30	40	50	60
	2	2	90.0	92.0	-	-	-	-		2	2	100.0	-	-	-	-	-
	2	3	90.0	92.0	-	-	-	-		2	3	100.0	-	-	-	-	-
flex	2	4	92.0	92.0	-	-	-	-	gzip	2	4	100.0	-	-	-	-	-
	2	5	90.0	90.0	-	-	-	-		2	5	100.0	-	-	-	-	-
	2	6	90.0	92.0	-	-	-	-		2	6	100.0	-	-	-	-	-
	2	2	-	-	-	-	-	-		2	2	-	-	-	-	-	-
	2	3	-	-	-	-	-	-		2	3	-	-	-	-	-	-
make	2	4	-	-	-	-	-	-	nanoxml	2	4	-	-	-	-	-	-
	2	5	-	-	-	-	-	-		2	5	-	-	-	-	-	-
	2	6	-	-	-	-	-	-		2	6	-	-	-	-	-	-
	2	2	75.0	75.0	83.33	91.67	-	-		2	2	100.0	-	-	-	-	-
	2	3	50.0	75.0	91.67	91.67	-	-		2	3	100.0	-	-	-	-	-
grep	2	4	91.67	91.67	91.67	91.67	-	-	siena	2	4	75.0	-	-	-	-	-
	2	5	83.33	83.33	91.67	91.67	-	-		2	5	50.0	-	-	-	-	-
	2	6	75.0	83.33	91.67	91.67	-	-		2	6	75.0	-	-	-	-	-

TABLE 22
ACTS Runtimes, Averages Over 20 Runs

CIT specification	Seconds $t=2$	Seconds $t=3$	Seconds $t=4$	Seconds $t=5$	Seconds t=6	Size $t=2$	Size $t=3$	Size $t=4$	Size $t=5$	Size t=6
FLEX constrained $CA(N;t,2^2 3^2 2^4 5^1)$	0.026	0.017	0.016	0.019	0.017	27	66	130	238	366
FLEX unconstrained $CA(N;t,2^{23} 3^4 5^2)$	0.003	0.014	0.222	6.029	162.402	26	97	361	1,171	3,547
MAKE constrained $CA(N;t,2^{10})$	0.008	0.007	0.007	0.006	0.007	9	19	43	91	185
MAKE unconstrained $CA(N;t,2^{14} 3^4 4^2 5^1 6^1)$	0.002	0.009	0.125	1.734	51.630	31	141	608	2,241	7,289
GREP constrained $CA(N;t,3^2 4^1 6^1 8^1 4^1 3^1 2^1 5^1)$	7.560	7.364	7.374	7.495	7.674	46	153	298	436	438
GREP unconstrained $CA(N;t,1^4 2^1 3^3 4^1 5^1 7^1 10^1 13^1 21^1)$	0.002	0.020	0.251	3.426	94.293	275	2,739	19,406	97,879	418,645
SED constrained $CA(N;t,2^4 6^1 10^1 2^1 4^1 2^2 3^1)$	0.622	0.561	0.571	0.611	0.721	58	170	324	324	324
SED unconstrained $CA(N;t,1^1 2^7 3^1 4^3 5^3 6^1 8^2 10^1)$	0.002	0.022	0.378	10.056	279.875	84	679	4,608	27,697	149,289
GZIP constrained $CA(N;t,2^{13} 3^1)$	0.096	0.046	0.042	0.070	0.132	21	68	108	144	144
GZIP unconstrained $CA(N;t,1^4 2^8 3^8 4^2 5^1 6^1 34^1)$	0.007	0.030	0.553	15.598	486.390	206	1094	5,284	2,3871	95,502
NANOXML constrained $CA(N;t,2^5 4^1 2^1)$	0.026	0.017	0.011	0.010	0.008	8	24	32	64	64
NANOXML unconstrained $CA(N;t,1^2 2^{11} 3^6 4^1 6^1)$	0.003	0.006	0.066	0.587	10.787	26	105	398	1,287	3,973
SIENA constrained $CA(N;t,3^1 4^1 3^1 5^1 4^2 3^3)$	0.187	0.193	0.157	0.174	0.209	22	83	216	428	516
SIENA unconstrained $CA(N;t,4^1 8^1 2^1 8^1 7^1 5^1 8^1 13^1 8^1 9^1 3^1)$	0.001	0.001	0.006	0.310	0.180	24	111	460	1,651	5,399

Constrained CAs were obtained by using the 'forbidden tuples' setting for constraint handing.

this potential threat to validity, we reported runtimes over 20 runs of CASA, ACTS and GAcit.

All of the programs we used in this study are relatively small with respect to the number of parameters. Moreover, there is little diversity in their type, four of them are text manipulation utilities.

But we have used real programs from a well-studied subject repository and believe that this represents a realistic use of CIT.

8 Conclusions

In this paper we investigated greedy, simulated annealing and genetic algorithm approaches to the constrained, prioritised, interaction testing problem, presenting results for their application to multiple versions of seven subjects using interaction strengths from two-way (pairwise) to six-way interactions. Our results hold for both C and Java programs used.

Our findings challenge the conventional wisdom that higher strength interaction testing is infeasible for simulated annealing; we were able to construct six-way interaction test suites in reasonable time. Furthermore, these higher strength test suites find more faults overall, making them worthwhile for comprehensive testing. We also find that ordering test suites for lower strengths performs no worse than higher strengths in terms of early fault revelation.

However, our findings also challenge the previously widely-held assumption that, compared to simulated

annealing, greedy algorithms are fast, yet produce larger test suites with lower fault revealing potency with respect to time. Not only did we find that, without careful selection of the constraint handling mechanism, greedy approaches can be surprisingly slow, but also more importantly, that their fault revealing power is comparable to that of simulated annealing. Genetic algorithms, on the other hand, do not scale to constrained higher-strength CIT.

Our results and test data, together with reports of coverage and fault detection and plots of Average Percentage of Covering-array Coverage for all cases are contained in this paper's companion website: http://www0.cs.ucl.ac.uk/staff/J.Petke/cittse/html/index.html.

Acknowledgments

Myra Cohen is partly supported by the National Science Foundation, through awards CCF-1161767, CCF-0747009 and by the Air Force Office of Scientific Research through award FA9550-10-1-0406. Mark Harman is partly supported by the following grants from the UK Engineering and Physical Sciences Research Council (EPSRC): DAASE: Dynamic Adaptive Automated Software Engineering, GISMO: Genetic Improvement of Software for Multiple Objectives and CREST: Centre for Research on Evolution, Search and Testing (Platform Grant). The DAASE grant also partly supports Shin Yoo, and completely supports Justyna Petke. The authors would also like to acknowledge the Software-artifact Infrastructure Repository (SIR) [20] which provided the source code and fault data for the seven programs used in the empirical studies reported. The authors would like to thank Tatsuhiro Tsuchiya for the GA tool used in this study and Wayne Motycka for advice on SIR. J. Petke is the corresponding author.

References

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

[2] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing," *Softw. Test., Verification Reliab.*, vol. 18, no. 3, pp. 125–148, 2008.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

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

[5] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, Jun. 2004.

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

[7] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 129–139.

[8] 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 Trans. Softw. Eng.*, vol. 34, no. 5, pp. 633–650, Sep./Oct. 2008.

[9] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2-3, pp. 199–219, 2006.

[10] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *Softw. Testing, Verification, Rel.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[11] R. C. Bryce and C. J. Colbourn, "Test prioritization for pairwise interaction coverage," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[12] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Trans. Softw. Eng.*, vol. 37, no. 1, pp. 48–64, Jan.-Feb. 2011.

[13] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2007, pp. 255–264.

[14] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proc. 28th Int. Comput. Softw. Appl. Conf., Des. Assessment Trustworthy Softw.-Based Syst.*, 2004, pp. 72–77.

[15] G. Dueck, "New optimization heuristics: The great deluge algorithm and the record-to-record travel," *J. Comput. Phys.*, vol. 104, no. 1, pp. 86–92, 1993.

[16] J. Stardom, *Metaheuristics and the Search for Covering and Packing Arrays* (Series Canadian theses). Thesis (M.Sc.)-Simon Fraser University, Burnaby, British Columbia, Canada, 2001.

[17] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Proc. Int. Conf. Softw. Eng.*, May 2003, pp. 38–48.

[18] D. Kuhn, R. Kacker, and Y. Lei, "Automated combinatorial test methods: Beyond pairwise testing," *Crosstalk, J. Defense Softw. Eng.*, vol. 21, no. 6, pp. 22–26, 2008.

[19] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng.*, 2014, pp. 323–334.

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

[21] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Saint Petersburg, Russian Federation, Aug. 2013, pp. 26–36.

[22] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proc. 30th Annu. IEEE/NASA Softw. Eng. Workshop*, 2006, pp. 153–158.

[23] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru, "Prioritizing user-session-based test cases for web applications testing," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2008, pp. 141–150.

[24] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 75–86.

[25] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Trans. Softw. Eng.*, vol. 31, no. 1, pp. 20–34, Jan. 2006.

[26] P. Schroeder, P. Bolaki, and V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," in *Proc. Empirical Softw. Eng.*, Aug. 2004, pp. 49–59.

[27] M. Grindal, J. Offutt, and J. Mellin, "Handling constraints in the input space when using combination strategies for software testing," Univ. of Skövde, Sweden, Tech. Rep. TR-06-001, 2006.

[28] T. Nanba, T. Tsuchiya, and T. Kikuno, "Using satisfiability solving for pairwise testing in the presence of constraints," *Inst. Electron., Inform. Commun. Eng. Trans.*, vol. 95-A, no. 9, pp. 1501–1505, 2012.

[29] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *Int. J. Syst. Assurance Eng. Manage.*, vol. 2, no. 2, pp. 126–134, 2011.

[30] X. Qu and M. B. Cohen, "A study in prioritization for higher strength combinatorial testing," in *Proc. 2nd Int. Workshop Combinatorial Testing*, 2013, pp. 285–294.

[31] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Proc. Workshop Domain Specific Approaches Softw. Test Autom.: In Conjunction with 6th ESEC/FSE Joint Meeting*, 2007, pp. 1–7.

[32] L. Shi, C. Nie, and B. Xu, "A software debugging method based on pairwise testing," in *Proc. Int. Conf. Comput. Sci.*, 2005, pp. 1088–1091.

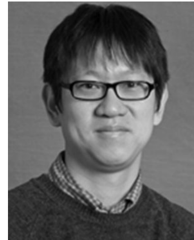
[33] M. Cohen, P. Gibbons, W. Mugridge, C. Colbourn, and J. Collofello, "A variable strength interaction testing of components," in *Proc. 27th Annu. Int. Comput. Softw. Appl. Conf.*, Nov. 2003, pp. 413–418.

[34] T. J. Ostrand and M. J. Balcer, “The category-partition method for specifying and generating functional tests,” *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.
[35] S. Manchester, N. Samant, R. Bryce, S. Sampath, D. R. Kuhn, and R. Kacker, “Applying higher strength combinatorial criteria to test prioritization: A case study,” *J. Combinatorial Math. Combinatorial Comput.*, vol. 86, pp. 51–72, Aug. 2013.
[36] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2000, pp. 102–112.
[37] S. Fouché, M. B. Cohen, and A. A. Porter, “Incremental covering array failure characterization in large configuration spaces,” in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 177–188.

Justyna Petke received the BSc degree in mathematics and computer science from the University College London, and the DPhil degree in computer science from the University of Oxford. She is a research associate at the Centre for Research on Evolution, Search and Testing (CREST), located in the Department of Computer Science, University College London. Her current research interests include genetic improvement, combinatorial interaction testing, and constraint solving.

Myra B. Cohen received the PhD degree from the University of Auckland, New Zealand, and the MS degree from the University of Vermont. She is a Susan J. Rosowski associate professor in computer science and engineering at the University of Nebraska-Lincoln, where she is a member of the ESQuaReD software engineering research group. Her research interests include software testing of highly configurable software, combinatorial interaction testing, testing of software product lines, and search-based software engineering. She received the US National Science Foundation Early Career award and an Air Force Office of Scientific Research Young Investigator award. She has served on the PCs of many software engineering conferences including ICSE, ISSTA, ESEC/FSE, ASE, and is the general chair of ASE 2015.

Mark Harman is the head of Software Systems Engineering and the director of the CREST at UCL. He is widely known for work on source code analysis and testing and was instrumental in the founding of the field of Search Based Software Engineering (SBSE), a sub- field of software engineering which is now attracted over 1,600 authors, spread over more than 40 countries.



Shin Yoo is a lecturer of software engineering at the Centre of Research on Evolution, Search and Testing (CREST), located in the Department of Computer Science, University College London. He has a broad range of research interests, but most of them are centred around search-based software engineering (SBSE), i.e., the application of meta-heuristic optimisation algorithms to problems in software engineering. His main research interest is regression testing and fault localisation, as well as how information theory and information retrieval techniques can be used to aid software testing process.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.