Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing

Justyna Petke
University College London
Gower Street, London
WC1E 6BT, UK
j.petke@ucl.ac.uk

Shin Yoo
University College London
Gower Street, London
WC1E 6BT, UK
shin.yoo@ucl.ac.uk

Myra B. Cohen
University of Nebraska-Lincoln
Lincoln
NE 68588-0115, USA
myra@cse.unl.edu

Mark Harman
University College London
Gower Street, London
WC1E 6BT, UK
mark.harman@ucl.ac.uk

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. However, in order to be practically applicable, it must be able to cater for soft and hard real-world constraints and should, ideally, report a test priority order that maximises earliest fault detection. We show that we can achieve the highest strength CIT in 5.65 minutes on average. This was previously thought to be too computationally expensive to be feasible. Furthermore, we show that higher strength suites find more faults, while prioritisations using lower strengths are no worse at achieving early fault revelation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Combinatorial Interaction Testing, Prioritisation, Empirical Studies, Software Testing

1. INTRODUCTION

Combinatorial interaction testing is increasingly important because of the increasing use of configurations as a basis for the deployment of systems [21]. 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 (CIT) has proved a useful technique for uncovering faults.

However, in most CIT applications, the problem domain is constrained: some interactions are simply infeasible due to these constraints [2, 8, 9, 14]. The nature and description of such constraints is highly domain specific, yet taking account of them is essential in order for CIT to be usable in practice. Any CIT approach that fails to take account of constraints will produce many test cases that are either unachievable in practice or which yield expensively misleading results (such as false positives).

Another type of constraint, often referred to as a soft constraint [2] 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. However, there has been little work on CIT with this type of soft constraint.

The order in which the test cases are applied to the system under test is also increasingly important for effective and practical testing, both in general [30] and for CIT [1, 4, 25]. In many testing scenarios, the number of test cases makes a naive 'test all' approach impractical. 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.

For CIT approaches to testing, it is known that higher-strength interactions can reveal faults left uncovered by lower strengths [17]. However, it is widely believed that only the lowest strength (pairwise interactions) can be covered in reasonable time; higher strengths, such as those up to 5- and 6-way feature interactions, have been considered infeasibly expensive, even though they may lead to improved fault revelation [17, 21].

In this paper we study practical lower *and* higher strength CIT approaches that take account of both real-world constraints and the necessary ordering required to prioritise test cases. We present results from empirical studies that report on the relationship between the achievement of lower and higher interaction strengths, and their ability to find faults for the constrained prioritised interaction problem. There has been little previous work on the relationship between

ESEC/FSE '13, August 18–26, 2013, Saint Petersburg, Russia

constrained interaction problems and fault revelation, particularly with respect to soft constraints, and none on the problem of ordering test cases for early fault revelation with respect to constrained higher strength interactions.

This paper addresses this important gap in the literature. We report on a series of empirical evaluations of constrained prioritised higher strength interaction testing on multiple versions of five programs from the Software-artifact Infrastructure Repository (SIR) [10]. Our results provide several findings that are important to the scientific development of interaction testing and also to practising testers.

The primary findings of the studies we report are:

1. We show that higher-strength CIT is feasible, confounding 'conventional wisdom'. This surprising result arises because of the role played by constraints. We report that, though they constrain the choice of test cases, these same constraints can make higher-strength CIT achievable in reasonable time.

2. We show that separate consideration of single and multi valued parameters leads to significant runtime improvements for prioritisation and interaction coverage.

3. 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 feasible and desirable.

4. 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.

Overall, taken together, our results are very promising for the future of CIT research and practice. Our results indicate that taking account of realistic testing scenarios (that are typically constrained and necessitate test case ordering) creates a problem that is amenable to high-strength CIT techniques.

This will be a welcome message to the research community, which has, hitherto, eschewed higher strength testing, believing it to be too expensive. For the practising tester, concerned with the problem of testing systems with large configuration spaces, our results are equally encouraging. They show that high-strength CIT is practical for comprehensive testing, yet lower strengths can be relied upon to quickly find the first faults.

2. BACKGROUND

2.1 Related Work

Combinatorial interaction testing (CIT) has been used successfully as a system level test method [6, 7, 16, 17, 24–27, 29]. 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 input (or configuration) space. Research has shown that we can achieve high fault detection rates given a small set of test cases [6, 17, 25, 29].

Many of the current research directions into this technique examine specialised problems such as the addition of constraints between parameter values [2, 8, 13, 20, 24], or re-ordering (prioritising) test suites to improve early coverage [4, 5, 24–26]. Other work has studied the impact of testing at increasing higher strengths ($t > 2$) [15, 23].

Table 1: Overview of literature on fault detection, prioritisation and constraints: extracted from [21].

Authors	Fault detect.	Prioritisation	Constraints
Bryce et al. [3]	✓	✓	✓
Cohen DM et al. [6]			✓
Cohen MB et al. [9]	✓	✓	✓
Grindal et al. [13]			✓
Kuhn et al. [17]	✓		
Nie et el. [28]	✓		
Schroeder et al. [27]			✓

In a recent survey by Nie et al. [21] CIT research is categorised by a taxonomy to show the areas of study. We have extracted data from this table for 3 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.

2.2 Preliminaries

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

$$CA(N; t, v_1^{k_1} v_2^{k_2} ... v_m^{k_m})$$

where N is the size of the array, t is its strength, sum of $k_1, ..., k_m$ 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 3 parameters, where the first and third parameter can take 4 values and the second one can only take 3 values. Then the problem can be formulated as: $CA(N; 2, 4^1 3^1 4^1)$.

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 8 test cases. The importance of constraints is evident even in this small example.

We differentiate between two types of constraints in this work: *hard* and *soft*, terms first proposed by Bryce and Colbourn [2]. 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. These constraints are combinations of parameters that we do not need to test 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 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 3-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 3-way, 4-way, or more (..). The few tools that do generate tests with interaction strengths higher than 2-way may require several days to generate tests (..) because the generation process is mathematically complex" [15]. However, recent work in this area shows a promising progress towards higher strength interaction coverage [12, 15, 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 covering arrays using state-of-the-art tools?

Greedy [6, 18] and meta-heuristic search [12] are the two most frequently used approaches for covering array generation [12]. Both involve a certain degree of randomness. For instance, simulated annealing, a meta-heuristic search technique, randomly selects a transformation, applies it, and compares 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 according to pairwise coverage has been found to be successful at finding faults quickly [5]. 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:

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	-	-

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?

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 [30]. The objective of TCP is to order tests in such a way that maximises the early detection of faults. This motivates our final research question:

RQ5: How effective are the prioritised test suites at detecting faults?

- 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 4-way constrained covering arrays and prioritising by pairwise coverage?

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. Their sizes in Uncommented Lines of Code, measured with `cloc`[1] are presented in Table 2. These are obtained from the Software-artifact Infrastructure Repository (SIR) [10].

We chose these 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 [23, 25]). In this previous work, the unconstrained versions of the subjects were used. Moreover, these five C subjects come with test plans described in the Test Suite Specification Language (TSL) [22].

We use TSL description to extract the relevant parameters and values and to define our constraints. Since TSL is created

[1] `http://cloc.sourceforge.net`

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.

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 would come from our making decisions about which constraints to retain or remove. For the generation of Covering Arrays, we have only considered parameters having at least two possible values[2] 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 the Covering Arrays by Simulated Annealing (CASA) tool[3] 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 [12].

Another reason to use CASA is to avoid one potential source of experimental bias. Most of the tools that are based on the 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 prefer an algorithm that does not implicitly perform prioritisation during its selection phase.

4.3 CA Prioritisation

After generating t-way covering arrays, we prioritise each of these according to multiple t-way prioritisation criteria (for $2 \leq t \leq 5$). There are standard prioritisation algorithms in the literature: Bryce and Memon [3], and Manchester et al. [19], for example[4].

For our experiments, we use a variation of the algorithm by Bryce and Memon [3]. We note that this differs from the code-coverage weighted prioritizaton of Qu et al. [25]. The original algorithm iterates through test cases and retains the one test case that covers the largest number of currently uncovered t-tuples.

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 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 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, and then pick one at random. 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. 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. 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.

```
CA = test suite to prioritize
gather all valid t-tuples based on CA
mapping=[]
sums=[]
for all tests in CA do
    mapping[test]=[True if t-tuple_i in test, else False]
    sums[test]=sum(mapping[test])
end for
bestTest = a test that covers the most unique t-tuples
add bestTest to TestSuite
selectedTestCount = 1
while selectedTestCount < size(CA) do
    update sums, mapping
    remove sums[bestTest], mapping[bestTest]
    tCountMax = max(sums)
    bestTests = []
    for all tests in sums do
        if sums[test] == tCountMax then
            add test to bestTests
        end if
    end for
    bestTest = random test from bestTests
    add bestTest to TestSuite
    selectedTestCount++
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% of all the parameters in the case of FLEX. Consequently, a lot of the same combinations of t-tuples are checked using Algorithm 2, even though many are already covered by the first test case selected. Therefore, we used the following combinatorial identity which improves efficiency by dividing effort between single valued and multi valued parameters:

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

where m and n are the numbers of single and multi valued parameters respectively and t is the interaction strength.

[2] 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.

[3] CASA is available at: `http://cse.unl.edu/~citportal/tools/casa/`.

[4] Note that the two algorithms differ only at the pre-processing stage.

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

```
CA= a given test suite
coverage=number of t-way interactions covered
coverage[0]=0
tuples=uncovered t-tuples
for j=1 to size(CA) do
  coverage[test_j]=coverage[test_{j-1}]
  for all t-tuples in test_j do
    if t-tuple in tuples then
      coverage[test_j] += 1
      remove t-tuple from tuples
    end if
  end for
end for
rate = coverage / number of all valid t-tuples * 100%
```

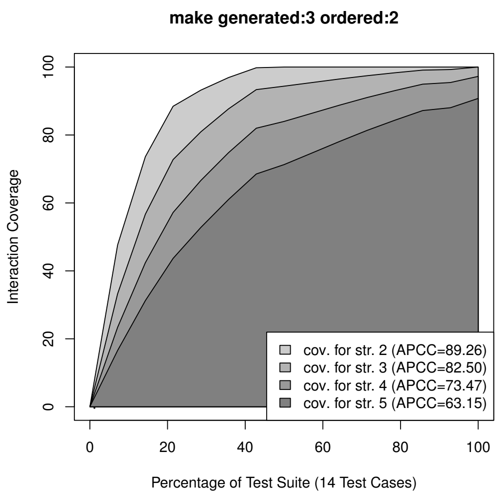


Figure 1: Interaction coverage of 3-way covering array for MAKE prioritised by pairwise coverage.

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 [11]. Given m covering arrays to cover and n test cases, let CA_i be the index of the first test case that covers the covering array CA. APCC is defined as follows:

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

APCC measures the area under curve for the plot of increasing interaction coverage for a prioritised test suite. Figure 1 illustrates the metric using the test suite generated for MAKE. It takes 14 test cases to achieve 100% coverage for 3-way interaction coverage. The test suite achieves 100% coverage for both 3-way and pairwise interaction coverage.

4.5 Fault Detection

We measure the fault detection capability of each prioritised test suite. We use all available software versions of the five 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. For each of the test suites we gathered the number of faults detected by every i tests.

Table 3: Unconstrained covering array sizes [23, 25].

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

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 subsection.

5.1 CA Generation Under Constraints

For FLEX, MAKE and GREP modified TSL descriptions were used by Qu et al. and Qu and Cohen [23, 25] 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" [25]. 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" [25]. The sizes of the covering arrays generated for these modified files are presented in Table 3. For FLEX and GREP, the numbers for $t = 4$ and $t = 5$ were not provided, most probably due to time restrictions of the CA generator used.

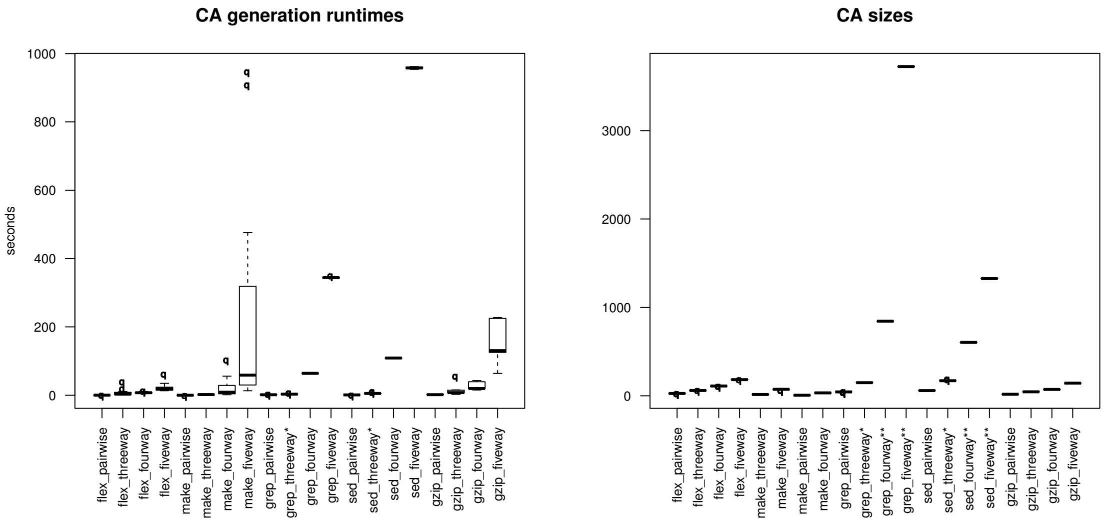
The sizes of the smallest constrained CA generated are presented in Table 4. In the case of GREP and MAKE for $t = 4$ and $t = 5$, only the numbers of unique rows are reported. The table also includes the number of tests in the original exhaustive TSL test suite from SIR. Tables 3 and 4 provide an answer to **RQ1**: constraints can reduce the size of CIT models significantly.

The constrained CIT models we use are generated directly from TSL descriptions from SIR and exclude the single valued parameters. We ran the CASA tool twenty times on each model on a Lenovo 3000 N200 laptop with an Intel Core 2 Duo processor, running at 1.66GHz with 2GB of RAM. Figure 2 presents the runtime information and the sizes of generated Covering Arrays.

Certain runs of CASA produced CAs with repeated rows (marked with ** in Figure 2). Most runs took fewer than 20 minutes. However, for the 3-way criterion of GREP and SED, CASA was terminated after an hour: subsequently, we ran CASA again, with the 'known size' parameter set to the best result obtained within an hour in these two cases. These runs are marked with * in Figure 2(a) and 2(b).

For comparison, we present the CIT models for the original TSL files from SIR with all the constraints and parameter order ignored in Table 5.

Results presented in this subsection 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 (printing the 'help' message, for example).



(a) CA Generation Runtime

(b) CA Sizes

Figure 2: Boxplots of CA Generation Runtimes and CA Sizes

Table 6: Interaction coverage for the five subjects tested.

Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5	Subjects	Gen. Crit.	Size	Cov. for Strength 2	3	4	5
`flex`	2	26	-	98.52	95.38	90.84	`grep`	4	356	-	-	-	99.71
	3	55	-	-	99.58	98.41		5	436	-	-	-	-
	4	111	-	-	-	99.93	`sed`	2	58	-	92.03	81.68	71.37
	5	180	-	-	-	-		3	170	-	-	98.85	96.48
`make`	2	7	-	94.60	84.47	71.88		4	324	-	-	-	100.00
	3	14	-	-	97.25	90.76		5	324	-	-	-	-
	4	30	-	-	-	98.94	`gzip`	2	18	-	97.56	93.00	87.08
	5	64	-	-	-	-		3	45	-	-	99.62	98.61
`grep`	2	43	-	88.71	74.07	59.92		4	72	-	-	-	99.95
	3	148	-	-	97.41	92.28		5	144	-	-	-	-

Table 4: Constrained CA sizes.

CIT specification	Size $t=2$	Size $t=3$	Size $t=4$	Size $t=5$	TSL full
FLEX					
$CA(N;t,2^2 3^2 2^4 5^1)$	26	55	111	180	525
MAKE					
$CA(N;t,2^{10})$	7	14	30	68	793
GREP					
$CA(N;t,3^2 4^1 6^1 8^1 4^2 3^1 2^1 5^1)$	43	148	356	436	470
SED					
$CA(N;t,2^4 6^1 10^1 2^1 4^1 2^2 3^1)$	58	170	324	324	360
GZIP					
$CA(N;t,2^{13} 3^1)$	18	45	72	144	214

Table 5: Constrained and unconstrained CIT models for subjects.

Constrained	Unconstrained
FLEX (32 TSL constraints)	
$CA(N;t,2^6 3^2 5^1)$	$CA(N;t,2^{23} 3^4 5^2)$
MAKE (28 TSL constraints)	
$CA(N;t,2^{10})$	$CA(N;t,2^{14} 3^4 4^2 5^1 6^1)$
GREP (58 TSL constraints)	
$CA(N;t,2^1 3^3 4^3 5^1 6^1 8^1)$	$CA(N;t,1^4 2^1 3^3 4^1 5^1 7^1 10^1 13^1 21^1)$
SED (58 TSL constraints)	
$CA(N;t,2^7 3^1 4^1 6^1 10^1)$	$CA(N;t,1^1 2^7 3^1 4^3 5^3 6^1 8^2 10^1)$
GZIP (69 TSL constraints)	
$CA(N;t,2^{13} 3^1)$	$CA(N;t,1^4 3^8 2^8 4^2 5^1 6^1 34^1)$

Table 7: **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.	$t = 2$	0.051
With single valued params.	$t = 2$	0.238
Without single valued params.	$t = 3$	0.097
With single valued params.	$t = 3$	18.249
Without single valued params.	$t = 4$	0.197
With single valued params.	$t = 4$	1079.809
Without single valued params.	$t = 5$	0.251
With single valued params.	$t = 5$	>20min

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

The significance of these reductions can be seen in Table 4. The number of test cases generated decreases significantly when compared to the full TSL suite. In the case of MAKE, 5-way coverage is achieved with only 68 tests, while the exhaustive test suite contains 793 test cases.

With regard 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.

However, all runs (including ones for higher strength CAs) finished in under 20 minutes, showing that state-of-the-art CA generation tools can cope high higher strength CA generation under constraints (**RQ2**). Unlike execution time, we observe little variance in CA sizes between the different runs of CASA (Figure 2(b)), providing an answer to **RQ3**. These two observations provide supporting evidence for the best practice, which is to perform a few runs of the tool with predetermined time-out and then to select the smallest CA generated.

5.2 Prioritisation and Interaction Coverage

This section addresses **RQ4**. Following the best practice outlined in Section 5.1, we chose 20 smallest Covering Arrays, out of the CAs we generated, for the combination of the subjects (FLEX, MAKE, GREP, SED and GZIP) and t-way interaction coverage criteria ($2 \leq t \leq 5$). Note that these only contain the multi-value parameters (single valued parameters having been removed). Subsequently, we ordered each of these according to pairwise, 3-way, 4-way and 5-way coverage using the greedy algorithm presented in Algorithm 1. This produces 80 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 7, the runtimes of Algorithm 1 for CIT models of FLEX with and without the single valued parameters.

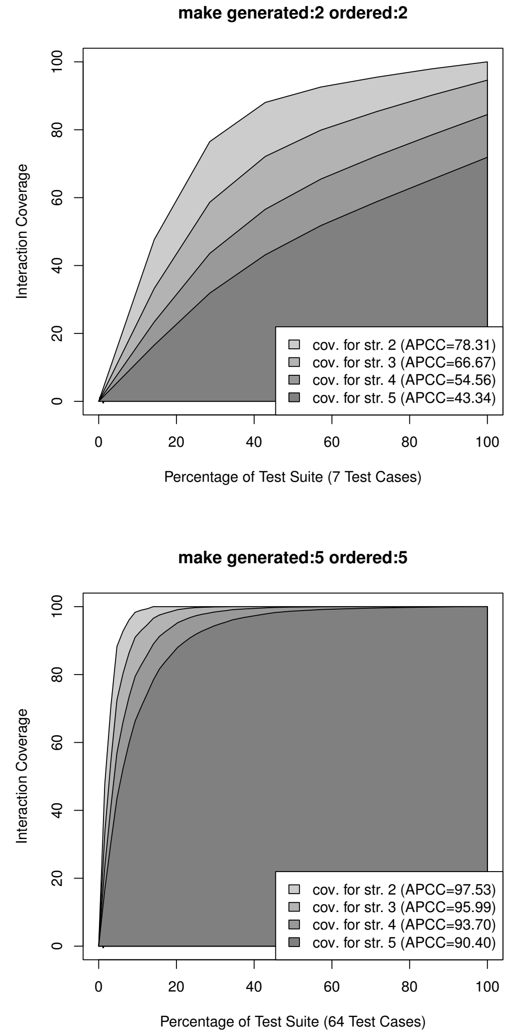


Figure 3: Comparing APCC for pairwise and 5-way CAs for MAKE.

Prioritising the same CA according to interaction coverage for different strengths produces significantly different permutations of test cases. Table 8 shows the permutations of the pairwise CA of MAKE according to different strength criteria.

Table 6 reports the interaction coverage achieved by each of the 80 CAs. For each CA generated for t-way strength, we measure the interaction coverage for t'-way strength ($2 \leq t' \leq 5$). A t-way strength CA, by definition, achieves 100% interaction coverage for strengths lower than t (therefore we omit these from Table 6).

For all subject programs, pairwise CAs achieve at least about 60% *collateral* 5-way interaction coverage. This provides an answer to the top level **RQ4**: the presence of constraints increases the collateral coverage for higher-strength interaction coverage. Note that, for coverage calculation, single valued parameters need to be added back to the CAs in order to produce full test suites.

To answer the subquestions of **RQ4** on prioritisation, we prioritised each of the 20 CAs according to four different prioritisation criteria (2-, 3-, 4-, and 5-way interaction cov-

Table 9: APCC values for the five subjects tested.

Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5	Subjects	Gen. Crit.	Prio. Crit.	APCC for Strength 2	3	4	5
`flex`	2	2	91.17	85.37	78.63	71.46	`grep`	4	2	97.69	93.67	88.34	82.43
	2	3	90.87	85.16	78.65	71.74		4	3	97.62	94.76	90.47	85.13
	2	4	90.32	84.55	78.05	71.19		4	4	97.36	94.55	90.75	86.10
	2	5	89.93	84.31	77.95	71.19		4	5	97.18	94.35	90.57	85.97
`flex`	3	2	95.95	92.83	88.76	83.94	`grep`	5	2	98.11	94.46	89.77	84.73
	3	3	95.71	92.75	89.08	84.77		5	3	97.99	95.70	92.18	87.74
	3	4	95.37	92.26	88.55	84.29		5	4	97.93	95.66	92.57	88.77
	3	5	94.73	91.57	87.88	83.70		5	5	97.67	95.43	92.41	88.71
`flex`	4	2	97.99	96.39	94.21	91.49	`sed`	2	2	85.86	72.17	59.34	48.58
	4	3	97.86	96.38	94.42	91.96		2	3	85.83	72.35	59.70	49.04
	4	4	97.72	96.17	94.20	91.81		2	4	85.40	71.94	59.40	48.85
	4	5	97.45	95.77	93.71	91.29		2	5	84.96	71.68	59.30	48.85
`flex`	5	2	98.70	97.68	96.22	94.35	`sed`	3	2	95.22	88.74	81.24	73.72
	5	3	98.66	97.77	96.57	95.00		3	3	95.18	89.51	82.74	75.65
	5	4	98.53	97.55	96.33	94.82		3	4	95.06	89.38	82.66	75.66
	5	5	98.38	97.36	96.10	94.58		3	5	94.92	89.34	82.72	75.78
`make`	2	2	78.31	66.67	54.56	43.34	`sed`	4	2	97.49	93.10	87.98	82.95
	2	3	78.51	67.42	55.51	44.25		4	3	97.50	94.59	90.85	86.63
	2	4	78.51	67.42	55.51	44.25		4	4	97.43	94.55	90.95	86.96
	2	5	78.51	67.42	55.51	44.25		4	5	97.35	94.48	90.90	86.95
`make`	3	2	89.26	82.50	73.47	63.15	`sed`	5	2	97.48	93.28	88.36	83.44
	3	3	89.20	82.72	74.01	63.87		5	3	97.47	94.58	90.83	86.59
	3	4	89.17	82.67	73.96	63.85		5	4	97.48	94.60	91.00	87.01
	3	5	88.65	82.24	73.67	63.66		5	5	97.40	94.52	90.95	87.02
`make`	4	2	95.04	91.59	86.56	79.87	`gzip`	2	2	83.44	75.32	67.33	59.79
	4	3	94.95	91.62	86.76	80.17		2	3	83.45	75.38	67.46	59.99
	4	4	94.95	91.83	87.21	80.86		2	4	83.37	75.38	67.51	60.07
	4	5	94.95	91.90	87.31	80.95		2	5	82.83	74.89	67.12	59.78
`make`	5	2	97.65	95.89	93.12	89.22	`gzip`	3	2	93.01	88.90	84.30	79.42
	5	3	97.61	96.05	93.61	90.04		3	3	93.10	89.09	84.61	79.83
	5	4	97.64	96.08	93.77	90.42		3	4	93.10	89.09	84.59	79.79
	5	5	97.53	95.99	93.70	90.40		3	5	93.08	89.04	84.52	79.71
`grep`	2	2	79.69	63.04	48.32	36.58	`gzip`	4	2	95.86	93.24	90.06	86.51
	2	3	78.76	62.76	48.33	36.67		4	3	95.86	93.36	90.34	86.96
	2	4	79.29	63.03	48.51	36.82		4	4	95.85	93.32	90.28	86.87
	2	5	78.21	62.49	48.28	36.74		4	5	95.85	93.32	90.28	86.87
`grep`	3	2	94.23	86.15	76.33	66.27	`gzip`	5	2	97.93	96.53	94.70	92.56
	3	3	93.98	87.17	78.36	68.77		5	3	97.93	96.69	95.15	93.34
	3	4	93.57	86.89	78.27	68.86		5	4	97.93	96.69	95.18	93.46
	3	5	93.49	86.74	78.17	68.87		5	5	97.93	96.68	95.16	93.41

Table 8: Permutations of the test suite for MAKE which achieves pairwise interaction coverage.

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

erage), resulting in 80 prioritised CAs. The results from the prioritisation are aggregated using APCC (defined in Section 4.4) in Table 9[5].

The variation in APCC between the different strengths of prioritisation criteria is observed to have very little effect. It caused less than 2% variation, and this variation decreases as the strength of the test suite increases, as can be seen in Figure 3.

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 3-way interaction to the test suite, it does not necessarily mean that a new pairwise interaction has been added.

However, whenever a new 2-way interaction is added, then automatically new 3-way, 4-way and 5-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.

5.3 Fault Detection

This section addresses **RQ5**. Table 10 presents the percentage of detected faults after 25%, 50%, 75%, and 100% of each test suite is executed, aggregated over all versions of subject programs. With FLEX, GREP, and SED, 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 covering 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 4-way covering arrays, for MAKE we just needed pairwise

[5] The complete data and all APCC plots are available at the companion webpage: `http://www0.cs.ucl.ac.uk/staff/s.yoo/cit/cit.html`.

Table 10: Percentage of detected faults for all versions of 5 subjects.

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	86.0	88.0	90.0	92.0	`flex`	4	2	94.0	94.0	98.0	100.0
	2	3	92.0	92.0	92.0	92.0		4	3	96.0	98.0	98.0	100.0
	2	4	66.0	90.0	90.0	92.0		4	4	92.0	96.0	98.0	100.0
	2	5	90.0	90.0	92.0	92.0		4	5	92.0	94.0	98.0	100.0
`flex`	3	2	94.0	94.0	94.0	94.0	`flex`	5	2	96.0	100.0	100.0	100.0
	3	3	90.0	94.0	94.0	94.0		5	3	96.0	100.0	100.0	100.0
	3	4	88.0	94.0	94.0	94.0		5	4	94.0	100.0	100.0	100.0
	3	5	74.0	92.0	94.0	94.0		5	5	94.0	94.0	98.0	100.0
`make`	2	2	50.0	100.0	100.0	100.0	`make`	4	2	100.0	100.0	100.0	100.0
	2	3	100.0	100.0	100.0	100.0		4	3	100.0	100.0	100.0	100.0
	2	4	100.0	100.0	100.0	100.0		4	4	100.0	100.0	100.0	100.0
	2	5	100.0	100.0	100.0	100.0		4	5	100.0	100.0	100.0	100.0
`make`	3	2	100.0	100.0	100.0	100.0	`make`	5	2	100.0	100.0	100.0	100.0
	3	3	100.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
	3	4	100.0	100.0	100.0	100.0		5	4	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0
`grep`	2	2	91.67	91.67	91.67	91.67	`grep`	4	2	91.67	100.0	100.0	100.0
	2	3	83.33	91.67	91.67	91.67		4	3	91.67	91.67	100.0	100.0
	2	4	83.33	83.33	83.33	91.67		4	4	100.0	100.0	100.0	100.0
	2	5	75.0	83.33	83.33	91.67		4	5	91.67	100.0	100.0	100.0
`grep`	3	2	91.67	91.67	100.0	100.0	`grep`	5	2	100.0	100.0	100.0	100.0
	3	3	91.67	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
	3	4	91.67	91.67	100.0	100.0		5	4	91.67	91.67	91.67	100.0
	3	5	83.33	91.67	100.0	100.0		5	5	91.67	100.0	100.0	100.0
`sed`	2	2	90.48	90.48	90.48	95.24	`sed`	4	2	85.71	95.24	100.0	100.0
	2	3	80.95	85.71	90.48	95.24		4	3	95.24	100.0	100.0	100.0
	2	4	85.71	95.24	95.24	95.24		4	4	95.24	95.24	100.0	100.0
	2	5	76.19	90.48	95.24	95.24		4	5	90.48	100.0	100.0	100.0
`sed`	3	2	85.71	90.48	95.24	100.0	`sed`	5	2	95.24	95.24	95.24	100.0
	3	3	90.48	100.0	100.0	100.0		5	3	95.24	100.0	100.0	100.0
	3	4	90.48	100.0	100.0	100.0		5	4	95.24	95.24	100.0	100.0
	3	5	90.48	90.48	95.24	100.0		5	5	85.71	100.0	100.0	100.0
`gzip`	2	2	80.0	100.0	100.0	100.0	`gzip`	4	2	100.0	100.0	100.0	100.0
	2	3	100.0	100.0	100.0	100.0		4	3	100.0	100.0	100.0	100.0
	2	4	80.0	100.0	100.0	100.0		4	4	100.0	100.0	100.0	100.0
	2	5	80.0	80.0	100.0	100.0		4	5	100.0	100.0	100.0	100.0
`gzip`	3	2	80.0	100.0	100.0	100.0	`gzip`	5	2	100.0	100.0	100.0	100.0
	3	3	80.0	100.0	100.0	100.0		5	3	100.0	100.0	100.0	100.0
	3	4	100.0	100.0	100.0	100.0		5	4	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0		5	5	100.0	100.0	100.0	100.0

coverage; for GREP, 3-way coverage; for SED, 3-way as well. For GZIP, it was sufficient to generate a pairwise covering array to detect the same faults as the full TSL suite.

Partially answering **RQ5**, we have found no 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 CAs. Table 11 presents the fault detection rate information against actual numbers of test cases executed, allowing direct comparison over all CAs: it shows the percentage of detected faults after multiples of 10 test case executions (CAs smaller than the given number of executions are marked with -). It provides a mixed response to the remainder of **RQ5**: there is no dominant prioritisation criterion with respect to fault detection rate after specific number of test executions, as 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 reliable fault detection rate.

6. CONCLUSIONS

In this paper we examined the constrained prioritised interaction testing problem for higher strengths, presenting results for multiple versions of five systems and interaction strengths from 2-way (pairwise) to 5-way interactions. Handling constraints and prioritisation are both important in order to make testing practical.

Real systems are typically constrained. Hard constraints must be accounted for to avoid the generation of inapplicable or misleading test cases, but soft constraints also can contribute to reducing the test space. Real testers also require prioritised set of test cases because they may not have time to simply apply all test cases available to them.

Therefore, to investigate these more practical forms of Combinatorial Interaction Testing, we report results for constrained prioritised interaction testing. More specifically, we study the relationship between interaction strength and faults found.

Table 11: Percentage of detected faults up to multiples of 10 test case executions.

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
flex	2	2	88.0	90.0	-	-	-	-	flex	4	2	86.0	94.0	94.0	94.0	94.0	98.0
	2	3	92.0	92.0	-	-	-	-		4	3	88.0	96.0	96.0	96.0	96.0	98.0
	2	4	70.0	90.0	-	-	-	-		4	4	88.0	92.0	92.0	92.0	92.0	96.0
	2	5	90.0	92.0	-	-	-	-		4	5	86.0	92.0	92.0	94.0	94.0	98.0
flex	3	2	92.0	94.0	94.0	94.0	94.0	-	flex	5	2	94.0	96.0	96.0	96.0	98.0	98.0
	3	3	90.0	94.0	94.0	94.0	94.0	-		5	3	88.0	90.0	92.0	96.0	96.0	96.0
	3	4	88.0	94.0	94.0	94.0	94.0	-		5	4	94.0	94.0	94.0	94.0	98.0	98.0
	3	5	72.0	74.0	92.0	94.0	94.0	-		5	5	92.0	92.0	94.0	94.0	94.0	94.0
make	2	2	-	-	-	-	-	-	make	4	2	100.0	100.0	-	-	-	-
	2	3	-	-	-	-	-	-		4	3	100.0	100.0	-	-	-	-
	2	4	-	-	-	-	-	-		4	4	100.0	100.0	-	-	-	-
	2	5	-	-	-	-	-	-		4	5	100.0	100.0	-	-	-	-
make	3	2	100.0	-	-	-	-	-	make	5	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	3	100.0	-	-	-	-	-		5	3	100.0	100.0	100.0	100.0	100.0	100.0
	3	4	100.0	-	-	-	-	-		5	4	100.0	100.0	100.0	100.0	100.0	100.0
	3	5	100.0	-	-	-	-	-		5	5	100.0	100.0	100.0	100.0	100.0	100.0
grep	2	2	91.67	91.67	91.67	91.67	-	-	grep	4	2	75.0	83.33	83.33	83.33	91.67	91.67
	2	3	83.33	83.33	91.67	91.67	-	-		4	3	75.0	83.33	91.67	91.67	91.67	91.67
	2	4	83.33	83.33	83.33	83.33	-	-		4	4	75.0	83.33	91.67	100.0	100.0	100.0
	2	5	75.0	83.33	83.33	83.33	-	-		4	5	83.33	83.33	91.67	91.67	91.67	91.67
grep	3	2	91.67	91.67	91.67	91.67	91.67	91.67	grep	5	2	75.0	83.33	83.33	83.33	83.33	91.67
	3	3	83.33	91.67	91.67	91.67	91.67	91.67		5	3	75.0	83.33	91.67	91.67	91.67	91.67
	3	4	83.33	91.67	91.67	91.67	91.67	91.67		5	4	83.33	91.67	91.67	91.67	91.67	91.67
	3	5	58.33	83.33	83.33	83.33	83.33	91.67		5	5	75.0	75.0	75.0	75.0	83.33	83.33
sed	2	2	80.95	90.48	90.48	90.48	90.48	-	sed	4	2	71.43	76.19	80.95	85.71	85.71	85.71
	2	3	61.9	85.71	85.71	90.48	95.24	-		4	3	61.9	80.95	90.48	95.24	95.24	95.24
	2	4	66.67	90.48	95.24	95.24	95.24	-		4	4	71.43	85.71	85.71	90.48	95.24	95.24
	2	5	61.9	85.71	90.48	90.48	95.24	-		4	5	71.43	71.43	90.48	90.48	90.48	90.48
sed	3	2	76.19	85.71	85.71	85.71	90.48	90.48	sed	5	2	66.67	95.24	95.24	95.24	95.24	95.24
	3	3	42.86	85.71	90.48	90.48	90.48	90.48		5	3	76.19	90.48	95.24	95.24	95.24	95.24
	3	4	80.95	85.71	85.71	90.48	95.24	95.24		5	4	80.95	80.95	85.71	90.48	95.24	95.24
	3	5	71.43	76.19	76.19	90.48	90.48	90.48		5	5	80.95	80.95	80.95	80.95	85.71	85.71
gzip	2	2	100.0	-	-	-	-	-	gzip	4	2	100.0	100.0	100.0	100.0	100.0	100.0
	2	3	100.0	-	-	-	-	-		4	3	100.0	100.0	100.0	100.0	100.0	100.0
	2	4	100.0	-	-	-	-	-		4	4	100.0	100.0	100.0	100.0	100.0	100.0
	2	5	80.0	-	-	-	-	-		4	5	100.0	100.0	100.0	100.0	100.0	100.0
gzip	3	2	80.0	100.0	100.0	100.0	-	-	gzip	5	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	3	80.0	100.0	100.0	100.0	-	-		5	3	100.0	100.0	100.0	100.0	100.0	100.0
	3	4	100.0	100.0	100.0	100.0	-	-		5	4	100.0	100.0	100.0	100.0	100.0	100.0
	3	5	100.0	100.0	100.0	100.0	-	-		5	5	100.0	100.0	100.0	100.0	100.0	100.0

The findings we report in this paper challenge the conventional wisdom that higher strength interaction testing is infeasible; we were able to construct 5-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.

We conclude that future work on interaction testing should exploit the largely untapped potential of higher strength test suites for comprehensive testing, but for ‘quick and usable’ results (seeking to find the first fault) we may be able to rely on lower strength prioritisation. To facilitate replication and to support this future work we make publicly available all data and results for our experiments.

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/s.yoo/cit/cit.html`.

7. ACKNOWLEDGEMENTS

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, RE-COST: REducing the Cost of Oracles for Software Testing and CREST: Centre for Research on Evolution, Search and Testing (Platform Grant). Grants RE-COST and DAASE also partly support both Shin Yoo, and completely support Justyna Petke.

The authors would also like to acknowledge the Software-artifact Infrastructure Repository (SIR) [10] which provided the source code and fault data for the five programs used in the empirical studies reported in this paper.

8. REFERENCES

[1] R. C. Bryce and C. J. Colbourn. Test prioritization for pairwise interaction coverage. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[2] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.

[3] R. C. Bryce and A. M. Memon. Test suite prioritization by interaction coverage. In A. Hartman, M. Katara, and A. M. Paradkar, editors, *DOSTA*, pages 1–7. ACM, 2007.

[4] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Trans. Software Eng.*, 37(1):48–64, 2011.

[5] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester. Test suite prioritization by cost-based combinatorial interaction coverage. *Int. J. Systems Assurance Engineering and Management*, 2(2):126–134, 2011.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[7] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48, May 2003.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In D. S. Rosenblum and S. G. Elbaum, editors, *ISSTA*, pages 129–139. ACM, 2007.

[9] 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. Software Eng.*, 34(5):633–650, 2008.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[11] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[12] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[13] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. Technical Report TR-06-001, University of Skövde, 2006.

[14] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[15] D. Kuhn, R. Kacker, and Y. Lei. Automated combinatorial test methods: Beyond pairwise testing. *Crosstalk, Journal of Defense Software Engineering*, 21(6), 2008.

[16] D. R. Kuhn and V. Okun. Pseudo-exhaustive testing for software. In *SEW*, pages 153–158. IEEE Computer Society, 2006.

[17] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

[18] 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., Verif. Reliab.*, 18(3):125–148, 2008.

[19] 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. *Journal of Combinatorial Mathematics and Combinatorial Computing*. To appear.

[20] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions*, 95-A(9):1501–1505, 2012.

[21] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11, 2011.

[22] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.

[23] X. Qu and M. B. Cohen. A study in prioritization for higher strength combinatorial testing. *The 2nd International Workshop on Combinatorial Testing*, 2013.

[24] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the International Symposium On Software Testing and Analysis*, pages 75–86, 2008.

[25] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *ICSM*, pages 255–264. IEEE, 2007.

[26] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru. Prioritizing user-session-based test cases for web applications testing. In *ICST*, pages 141–150. IEEE Computer Society, 2008.

[27] P. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of Empirical Software Engineering*, pages 49–59, August 2004.

[28] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *International Conference on Computational Science (3)*, pages 1088–1091, 2005.

[29] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

[30] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 22(2):67–120, March 2012.