ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 7, July 2016

C3 & LCOM based Combinational Interaction Testing for Java Application

Dr.K.V.Naganjaneyulu[1], P.Viswanatha Reddy [2], P.Jabeen Taj[3]

Professor & HOD, Dept. of CSE., Sir Vishveshwaraiah Institute Of Science &Technology, Madanapalle, AP, India.[1]

Assistant Professor, Dept. of CSE., Sir Vishveshwaraiah Institute Of Science &Technology, Madanapalle, AP, India.[2]

M.Tech Scholar, Dept. of CSE., Sir Vishveshwaraiah Institute Of Science &Technology, Madanapalle, AP, India.[3]

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 7 real-world subjects for the well-studied 2-way up to the rarely-studied 6-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 valuated 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 pair wise testing for subjects with a small number of constraints.

KEYWORDS**:** Testing, CIT, LCOM, Cohesion, GREEDY & SA

I. INTRODUCTION

In this paper we present the results of an empirical study of practical Combinatorial Interaction Testing. 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 (CIT) 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 2-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 6-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 5- and 6-way feature interactions, have been considered in feasibly 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). 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 prioritization 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 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 tradeoffs 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 pair wise 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 3-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. This paper addresses this important gap in the literature. We report on constrained, prioritised, CIT using Simulated Annealing, Greedy and Genetic Algorithms approaches for 2-way to 6-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 2-way to 5-way interaction strengths applied to 5 of the 7 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 6-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 prioritized 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 prioritization 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.

II. Related work

Combinatorial interaction testing (CIT) has been used successfully as a system level test method [3], [5], [13], [22], [17], [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 configurations) 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 toolis 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. Grinder et al. [27] evaluated seven constraint handling methods for test case selection. Bryce et al. 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 specialized problems such as the addition of constraints between parameter values [6], [7], [24], [28], [27], or re-ordering (prioritizing) 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 et al. [1] CIT research is categorized by taxonomy to show the areas of study. We have extracted data from this table for 3 columns, fault detection, constraints and prioritization. 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 prioritization, constraints and fault detection.

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 [33]: CA(N; t; vk1 1 vk22 :::vkmm) where N is the size of the array, t is its strength, sumof k1; :::; km is the number of parameters and each vi stands for the number of values for each of the ki

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 7, July 2016

parameters in turn. Suppose we want to generate a pair wise 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. The problem can then be formulated as: CA(N; 2; 413141), 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, pair wise 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.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 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.

III. PROPOSED ALGORITHM

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 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. 1-way interaction test suite will contain 3 test cases. If we add a constraint (p1 6= 2 ^ p2 6= 2) ! p3 = 2, we increase 1-way interaction test suite size to 4. strengths higher than 2-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 maximizes 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
4-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.)

IV. Pseudo code

```
Formula for Lcom5

Step1:  lcom=((((1/a)*Mu)-m)/deno);
Step2: Mu-count for fields , m-methods length, a-fields length, deno=1-m;
Step3: lcom51=lcom*lcom;
Step4: lcom5=Math.sqrt(lcom51);

If a=6,m=5, Mu=5 ,deno=4;
Locm5=0.95
```

V. Simulation Results

Proposals of measures and metrics for cohesion abound in the literature as software cohesion metrics proved to be useful in different tasks, including the assessment of design quality, productivity, design, and reuse effort, prediction of software quality, fault prediction modularization of software and identification of reusable of components.

Most approaches to cohesion measurement have automation as one of their goals as it is impractical to manually measure the cohesion of classes in large systems. The tradeoff is that such measures deal with information that can be automatically extracted from software and analyzed by automated tools and ignore less structured but rich information from the software (for example, textual information). Cohesion is usually measured on structural information extracted solely from the source code (for example, attribute references in methods and method calls) that captures the degree to which the elements of a class belong together from a structural point of view. These measures give information about the way a class is built and how its instances work together to address the goals of their design. The principle behind this class of metrics is to measure the coupling between the methods of a class. Thus, they give no clues as to whether the class is cohesive from a conceptual point of view (for example, whether a class implements one or more domain concepts) nor do they give an indication about the readability and comprehensibility of the source code. Although other types of metrics were proposed by researchers to capture different aspects of cohesion, only a few metrics address the conceptual and textual aspects of cohesion.

We propose a new measure for class cohesion, named the Conceptual Cohesion of Classes (C3), which captures the conceptual aspects of class cohesion, as it measures how strongly the methods of a class relate to each other conceptually. The conceptual relation between methods is based on the principle of textual coherence. We interpret the implementation of methods as elements of discourse. There are many aspects of a discourse that contribute to coherence, including co reference, causal relationships, connectives, and signals. The source code is far from a natural language and many aspects of natural language discourse do not exist in the source code or need to be redefined. The rules of discourse are also different from the natural language.

C3 is based on the analysis of textual information in the source code, expressed in comments and identifiers. Once again, this part of the source code, although closer to natural language, is still different from it. Thus, using classic natural language processing methods, such as propositional analysis, is impractical or unfeasible. Hence, we use an Information Retrieval (IR) technique, namely, Latent Semantic Indexing (LSI), to extract, represent, and analyze the textual information from the source code. Our measure of cohesion can be interpreted as a measure of the textual coherence of a class within the context of the entire system.

Cohesion ultimately affects the comprehensibility of source code. For the source code to be easy to understand, it has to have a clear implementation logic (that is, design) and it has to be easy to read (that is, good language use). These two properties are captured by the structural and conceptual cohesion metrics, respectively.

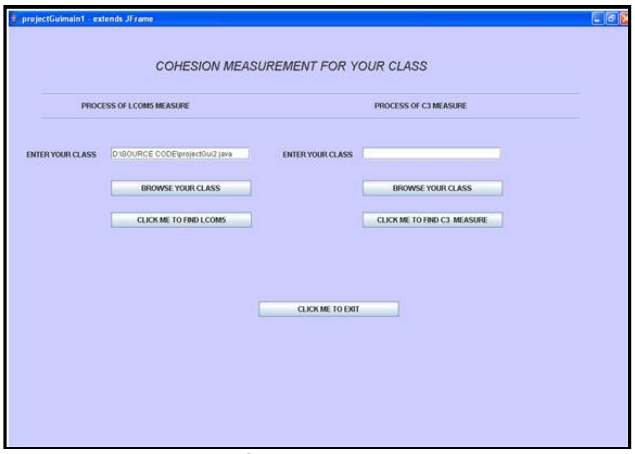
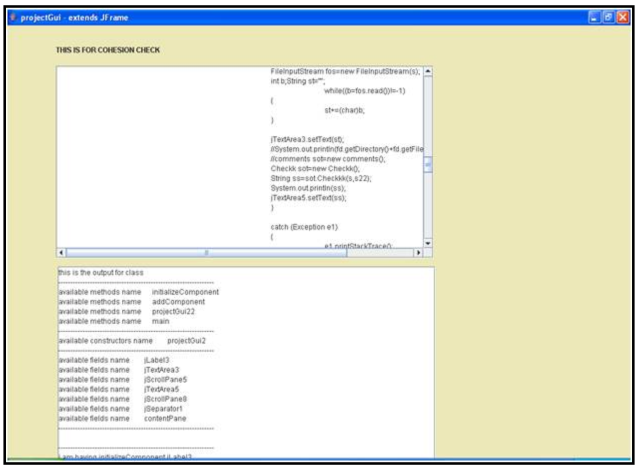


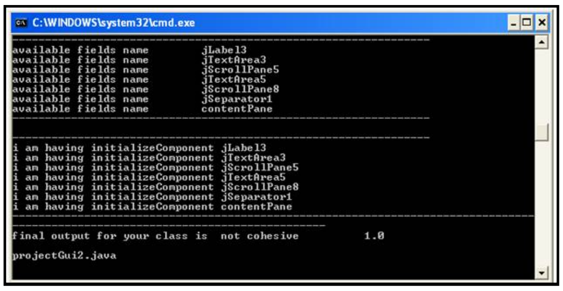
Fig.1. Home Screen



```
FileInputStream fos=new FileInputStream(s);
int b;String st="";
                    while((b=fos.read())!=-1)
{
                    st+=(char)b;
}

jTextArea3.setText(st);
//System.out.println(fd.getDirectory()+fd.getFile
//comments sot=new comments();
Checkk sot=new Checkk();
String ss=sot.Checkkk(s,s22);
System.out.println(ss);
jTextArea5.setText(ss);
}

catch (Exception e1)
{
                    e1.printStackTrace();
```

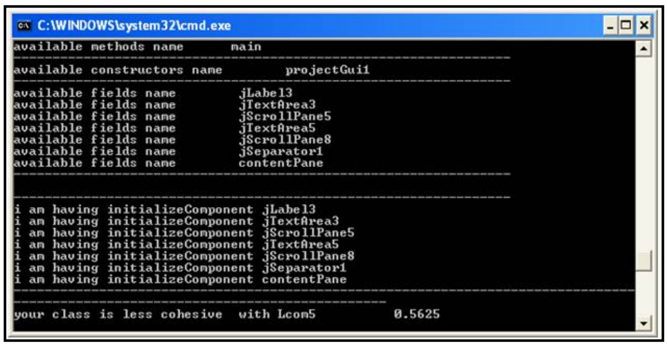
```
this is the output for class
------------------------------------------
available methods name     initializeComponent
available methods name     addComponent
available methods name     projectGui22
available methods name     main
------------------------------------------
available constructors name     projectGui2
------------------------------------------
available fields name     jLabel3
available fields name     jTextArea3
available fields name     jScrollPane5
available fields name     jTextArea5
available fields name     jScrollPane8
available fields name     jSeparator1
available fields name     contentPane
------------------------------------------

------------------------------------------
i am having initializeComponent jLabel3
```

Fig. 2. Results with Cohesion



```
------------------------------------------------------------
available fields name         jLabel3
available fields name         jTextArea3
available fields name         jScrollPane5
available fields name         jTextArea5
available fields name         jScrollPane8
available fields name         jSeparator1
available fields name         contentPane
------------------------------------------------------------

------------------------------------------------------------
i am having initializeComponent jLabel3
i am having initializeComponent jTextArea3
i am having initializeComponent jScrollPane5
i am having initializeComponent jTextArea5
i am having initializeComponent jScrollPane8
i am having initializeComponent jSeparator1
i am having initializeComponent contentPane
------------------------------------------------------------
final output for your class is  not cohesive          1.0

projectGui2.java
```

Fig. 3. Command Line Result for C3



```
available methods name        main
------------------------------------------------------------
available constructors name        projectGui1
------------------------------------------------------------
available fields name         jLabel3
available fields name         jTextArea3
available fields name         jScrollPane5
available fields name         jTextArea5
available fields name         jScrollPane8
available fields name         jSeparator1
available fields name         contentPane
------------------------------------------------------------

------------------------------------------------------------
i am having initializeComponent jLabel3
i am having initializeComponent jTextArea3
i am having initializeComponent jScrollPane5
i am having initializeComponent jTextArea5
i am having initializeComponent jScrollPane8
i am having initializeComponent jSeparator1
i am having initializeComponent contentPane
------------------------------------------------------------
your class is less cohesive  with Lcom5          0.5625
```

Fig 4. Command Line Result for Lcom

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 7, July 2016

VI. CONCLUSION AND FUTURE WORK

In this paper we investigated greedy, simulated annealing and genetic algorithm approaches to the constrained, prioritized, interaction testing problem, presenting results for their application to multiple versions of seven subjects using interaction strengths from 2-way (pair wise) to 6-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 6- 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.

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, specializing in various types of constraints) provides substantial overhead. For GREP result for 6-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..

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., Verif. 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 Transactions on Software Engineering, vol. 23, no. 7, pp. 437–444, 1997.
4. B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," Empirical Software Engineering, vol. 16, no. 1, pp. 61–102, 2011.
5. D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," IEEE Trans. Software Eng., vol. 30, no. 6, pp. 418–421, 2004.
6. R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," Information & Software Technology, 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 ISSTA, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007,pp. 129–139.
8. , "Constructing interaction test suites for highly configurable systems in the presence of constraints: A greedy approach," IEEE Trans. Software Eng., vol. 34, no. 5, pp. 633– 650, 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," Software Testing, Verification, and Reliability, vol. 22, no. 2, pp. 67–120, March 2012.
11. R. C. Bryce and C. J. Colbourn, "Test prioritization for pairwise interaction coverage," ACM SIGSOFT Software Engineering 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. Software Eng., vol. 37, no. 1, pp. 48–64, 2011.
13. X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in ICSM. IEEE, 2007, pp. 255–264.
14. T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in 28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings, 2004, pp. 72–77.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 7, July 2016

15. G. Dueck, "New optimization heuristics: The great deluge algorithm and the record-to-record travel," Journal of Computational Physics, vol. 104, no. 1, pp. 86 – 92, 1993.
16. J. Stardom, Metaheuristics and the Search for Covering and Packing Arrays, ser. Canadian theses. Thesis (M.Sc.)–Simon Fraser University, 2001.
17. 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, May 2003, pp. 38–48..
18. D. Kuhn, R. Kacker, and Y. Lei, "Automated combinatorial test methods: Beyond pairwise testing," Crosstalk, Journal of Defense Software Engineering, vol. 21, no. 6, pp. 22–26, 2008.
19. S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014. IEEE, 2014, pp. 323–334. [Online]. Available: http://dx.doi. org/10.1109/ISSRE.2014.15
20. H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Software Engineering, vol. 10, no. 4, pp. 405–435, 2005.
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 European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13. Saint Petersburg, Russian Federation: ACM, August 2013, pp. 26–36.
22. D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in SEW. IEEE Computer Society, 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 ICST. IEEE Computer Society, 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 Proceedings of the International Symposium On Software Testing and Analysis, 2008, pp. 75–86.
25. C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," IEEE Transactions on Software Engineering, vol. 31, no. 1, pp. 20–34, January 2006.