On The Effectiveness of Combinatorial Interaction Testing: A Case Study

Miroslav Bures
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Praha 2
Czech Republic
Email: buresm3@fel.cvut.cz

Bestoun S. Ahmed
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Praha 2
Czech Republic
Email: albeybes@fel.cvut.cz

***Abstract*—Combinatorial interaction testing (CIT) stands as one of the efficient testing techniques that have been used in different applications recently. The technique is useful when there is a need to take the interaction of input parameters into consideration for testing a system. The key insight the technique is that not every single parameter may contribute to the failure of the system and there could be interactions among these parameters. Hence, there must be combinations of these input parameters based on the interaction strength. This technique has been used in many applications to assess its effectiveness. In this paper, we are addressing the effectiveness of CIT for a real-world case study using model-based mutation testing experiments. The contribution of the paper is threefold: First we introduce an effective testing application for CIT; Second, we address the effectiveness of increasing the interaction strength beyond the pairwise (i.e., interaction of more than two parameters); Third, model-based mutation testing is used to mutate the input model of the program in contrast to the traditional code-based mutation testing process. Experimental results showed that CIT is an effective testing technique for this kind of application. In addition, the results also showed the usefulness of model-based mutation testing to assess CIT applications. For the subject of this case study, the results also indicate that $3-way$ test suite (i.e., interaction of three parameters) could detect new faults that can not be detected by pairwise.**

I. Introduction

In software systems, the interaction of input parameters is strong potential source of failure. Most of the test design techniques are considered input parameters individually for fault detection of a software-under-test (SUT). However, evidence showed that interaction among input parameters is a strong source of failure[1], [2], [3]. Combinatorial Interaction Testing (CIT) (sometimes called $t-way$ or $t-wise$ testing where t is the interaction strength) is an approach to overcome this shortage in the test design methods. CIT takes the interaction of two or more inputs in a generated test suite to help early fault detection in the testing life cycle [4]. The key insight of this testing approach is that, not every input parameter of the system contributes to the faults of the system and most of the faults are addressed by including interactions of only a few number of input parameters. To this end, the CIT is also useful to minimize the size of test suites by reducing the number of test cases and thus to prevent from the exhaustive testing as it is impractical and often also impossible from project resources viewpoint[5].

For instance, pairwise testing (i.e., $2-way$ testing) is a common approach and has been applied effectively in many practical testing situations [6]. However, empirical evidence have shown that software failures may be triggered by more than two input parameters and values [4]. Such cases have been identified as $t-way$ CIT where $t>2$.

Different strategies have been developed to generate $t-way$ test suites. In fact, some strategies have tried to generate test suites for high interaction strengths like $t=12$ [7]. However, these strategies faced two main criteria, efficiency, and effectiveness. Efficiency is characterized by the size of the generated test suites as compared to other strategies, whereas, the effectiveness is characterized by the applicability of the generated test suites [8]. In this paper, we are focusing on the second criterion.

In general, effectiveness deals with the applicability of the generated test suite by the CIT strategy. Within this context, there are also remarkable issues. In this paper, we are trying to address three main issues. First, we are seeking to address a new testing application of CIT approach, that shows the effectiveness of the generated test suites. Second, we are seeking to show the usefulness of those test suites beyond pairwise testing. Third, we are trying to show the usefulness of model-based mutation testing to assess the effectiveness of CIT by following the concepts of code-based mutation testing. We have used a well-known open source software as a subject of our case study. Model-based mutation testing is used within the case study as a framework to assess the effectiveness. We injected the software with different mutant based on the input models and then tried to detect those mutants by generated $t-way$ test suites. Here, mutation testing concepts is used to mutate the input models of the input parameters and then we tried to measure the number of detected mutated inputs.

The rest of this paper is organized as follows. Section II gives preliminaries and necessary mathematical backgrounds about CIT. Section III shows the related works achieved so far in this direction. Section IV illustrates the experimental setup, and Section V shows and discusses the results of the

experiments. Section VI presents and discusses the threats that may affect the validity of the experiments. Finally, Section VII gives the concluding remarks of the paper.

II. Combinatorial Interaction Testing (CIT)

Combinatorial Covering Array (CA) is a mathematical object that is used as a base in CIT. Originally, CA has been gained more attention as a practical alternative of oldest mathematical object called Orthogonal Array (OA) that has been used for statistical experiments [9]. An $OA_\lambda(N; t, k, v)$ is an $N \times k$ array, where for every $N \times t$ sub-array, each $t - tuple$ occurs exactly λ times, where $\lambda = N/v^t$; t is the combination strength; k is the number of input functions ($k \geq t$); and v is the number of values associated with each input parameter [10]. Fig. 1(a) illustrates an orthogonal array $OA(9; 2, 4, 3)$ that contains value ($v = 3$), with interaction degree ($t = 2$) , and input factors ($k = 4$) can be generated by nine rows. For real applications, practically, it is very hard to translate these firm rules except for small systems with small number of input parameters and values. Hence, there is no significant benefit in case of medium and large size systems, as it is very hard to generate OA for them. In addition, based on the aforementioned rules, it is not possible to represent OA when there are different levels for each input parameters.

To address the limitations of OA, CA has been introduced. A $CA_\lambda(N; t, k, v)$ is an $N \times k$ array over $(0, ..., v-1)$ such that every $N \times t$ sub-array contains all ordered subsets from v values of size t at least λ times, where the set of column $B = \{b_0, ..., b_{t-1}\} \supseteq \{0, ..., k-1\}$ [2]. In this case, each $t - tuple$ is to appear at least once in a CA. Fig. 1(b) shows an example of CA with $N = 9$, $k = 4$, $v = 3$, and $t = 2$.

In the case when the number of component values varies, this can be handled by Mixed Covering Array (MCA). A $MCA(N; t, k, (v_1, v_2, \dots v_k))$, is an $N \times k$ array on v values, where the rows of each $N \times t$ sub-array cover and all $t-tuples$ of values from the t columns occur at least once. For more flexibility in the notation, the array can be presented by $MCA(N; t, v_1^{k_1} v_2^{k_2} .. v_k^k)$. Fig.1(c) shows a MCA with size 9 that has four input parameters, with two input parameters having three values each and the other two input parameters having two values each.

III. Related Works

CA has been used to solve many complex problems. It has been used widely in different applications such as hardware testing [11], advance material testing [12], gene expression regulation [13], performance evaluation of communication systems [14] and many other applications [15]. It has also been used as an essential testing framework in many software applications and software product lines (SPL) (e.g., [16], [17], [18], [19], [20]). More recently, we have also discovered many applications for CA such as optimization of dynamic voltage scaling (DVS) in high performance processors [21], direct current (DC) servomotor controller [22], tuning of functional order PID controller [23], fault detection of software systems [24], [2], [25], graphical user interface (GUI) testing [26].

In fact, CA has been used wider in software testing activities for CIT. Bryce and Colbourn [27] introduced the prioritization concept within the CA to prioritize the test suites for regression testing. The key idea here is that the chance of repeated faults is frequent as the produced software is developed or maintained. Developers found that it is better to keep the test suites generated for the earlier version of the software and prioritize them based on specific mechanism. Hence, prioritization in this way increases the effectiveness of the test suites. Qu et al. [28] applied prioritization within CA successfully on two software subjects to examine the effectiveness of CA to find faults in the regression testing process. The results of the research showed that most of the faults can be found when $t = 2$ and 3.

CA has also been used to find faults' location in SUT within an approach of testing called fault characterization or failure diagnosis. Here, CIT is effective when the configuration space of the system is large. It helps to fix the faults quickly and saves a significant amount of time. Yilmaz et al.[29] applied CA for fault characterization on a software called "Skoll" that is used for distributed continuous quality assurance. The research found that even low interaction strengths can detect and localize faults effectively.

CIT is used widely with mutation analysis effectively. In fact, mutation testing is used within other testing approached as a powerful testing technique (see [30], [31]). Code-based mutation testing is used during the debugging process to evaluate fault detection capability of the test cases and thus identify the quality of the generated test suites [30]. Following this approach, few studies have used mutation testing to assess the quality of the test cases generated by CIT strategies (e.g., [32], [19], [18]). More recently, Belli et al. [33] have demonstrated the usefulness and the position of model-based mutation testing in the landscape of mutation testing. Here, model-based mutation testing is following the same concepts of traditional code-based mutation testing. However, the mutants are affecting the input model of the SUT. Contrary to those above-mentioned works in mutation testing, in this research, we are also trying to investigate the effectiveness of CIT using model-based mutation testing as a framework. Here, using model-based mutation testing concepts, we can know the quality of the used test cases and thus investigate the effectiveness. The following sections illustrate the experimental setup and the method of assessment.

IV. Experimental Setup

As mentioned previously, we aim to introduce a new application of CIT and know the effectiveness of combinatorial interaction test suites. In doing so, we will be able to reach our second aim to understand the effectiveness of $t - way$ test suites. In our experiments, we measured the effectiveness of $t - way$ combinatorial test suites for a defined business case, which was reporting an issue in an issue-tracking system (i.e., SUT). We used the business cases to model the inputs of the issue-tracking system. We then mutate those input models and run the experiments mainly to detect defects in the SUT. To

OA (9; 2, 4, 3)			
k_1	k_2	k_3	k_4
1	1	1	1
2	2	2	1
3	3	3	1
1	2	3	2
2	3	1	2
3	1	2	2
1	3	2	3
2	1	3	3
3	2	1	3

(a)

CA (9; 2, 4, 3)			
k_1	k_2	k_3	k_4
1	3	3	3
3	2	3	1
1	1	2	1
1	2	1	2
3	1	1	3
2	1	3	2
3	3	2	2
2	3	1	1
2	2	2	3

(b)

MCA (9; 2, 4, 3^2 2^2)			
k_1	k_2	k_3	k_4
2	1	1	2
2	2	2	1
3	3	2	2
1	3	1	1
1	1	2	1
1	2	1	2
3	2	1	1
3	1	1	1
2	3	1	2

(c)

Fig. 1. Examples illustrating OA, CA, and MCA [2]

simulate the defects, we prepared ten instances of the SUT, to which artificial defects were injected. We choose to use the SUT for the software development case. Here, base on our industrial experience, we manually mutate the inputs just like the standard code mutation in the code-based mutation testing. For each of these instances, we analyzed how effective are the individual test cases concerning the ability to detect the injected defects. The details of the experimental setup and the mutation process are given in the following sub-sections.

A. Object of Experiment

As SUT, we have selected the open-source JTrac[1] application, that is written in J2EE. The JTrac is an issue tracking system with configurable data fields and issues workflow. It uses a relational database for data storage with standard HTML and AJAX user interface. In our experiment, we focused on the part of entering the issue and immediate follow-up processing of the entered data. We enriched the data processing logic in the JTrac by more complex verification of the issue data, entered into the system.

We modeled an issue entry form for 9 fields of configuration, which are subjects of our interest. In each of the fields, we identified equivalence classes (ECs), as in a standard industrial test design process. The configuration of the issue entry form is presented in Table I.

Then, we defined several combinations of values, which are not allowed in the issue tracking process. At this point, we copied similar setup from a recent industrial project we have been consulting. We extended that part of the JTrac which is responsible for data verification by a set of conditions ensuring these constraints. We extended the system by 26 conditions in total. Out of these, 16 conditions were guarding particular combinations of 2 input fields, and 10 conditions were guarding particular combinations of 3 input fields.

[1]JTrac Official Web Page, http://jtrac.info/

An example of the 2 input fields condition is: IF ((operational_system==iOS) AND (browse==Edge)) THEN the combination is not valid.

An example of the 3 input fields condition is: IF ((issue_severity==1) AND (issue_priority==1) AND (issue_preliminary_classification==usabiliy_defect)) THEN the combination is not valid.

This was the baseline SUT, which was considered as the correct version. We used this baseline input also for implementation of the test oracle, determining the expected correct results of the tests.

B. Model-based Mutation Testing Process

We used the input model described in Section IV-A within a model-based mutation framework. Just like traditional code-based mutation testing, here, we prepared ten instances of baseline SUTs (mutated SUTs further on), to which we inserted different sets of artificial defects by standard code mutation techniques. We mutated the code affecting processing of the issue data after its input to the system via the form for entering issues. In this experiment, we focused on mutation of conditions f the input models. More details about mutated SUTs are given in Table II. It is worthy here to mention that we are not dealing with the code of JTrac.

In Table II, Column MUT-ALL gives a total number of mutated conditions in the SUT model. Column MUT-2 gives the number of mutated conditions, where two variables considered for inputs to the decision, whereas column MUT-3 gives the number of mutated conditions, where three variables considered for inputs to the decision. Moreover, columns “VALUE”, “AND” and “NOT” are giving the numbers of particular input mutation type, which was made in the SUT input model. These input mutation types are explained in Table III, using pseudo-code notations. Here, x stands for variable, N and M for constants. The input elements presented in Table III can be part of more complex decision expressions.

TABLE I
CONFIGURATION OF ISSUE ENTRY FORM

Field	Type	Number of ECs
Issue name	Short text	5
Issue description	Long text	4
Issue severity	List of values	5
Issue priority	List of values	5
Operational system	List of values	5
Browser	List of values	6
Testing environment	List of values	5
System	List of values	7
Issue preliminary classification	List of values	5

TABLE II
PROPERTIES OF MUTATED SUTS

Mutated SUT ID	MUT-ALL	MUT-2	MUT-3	VALUE	AND	NOT
1	1	0	1	0	0	1
2	1	0	1	1	0	0
3	1	0	1	1	0	0
4	2	0	2	2	0	0
5	3	1	2	3	0	0
6	4	1	3	2	2	0
7	5	2	3	3	2	0
8	6	2	4	5	0	1
9	7	3	4	3	0	4
10	8	4	4	3	1	4

TABLE III
INPUT MUTATION TYPES USED IN THE EXPERIMENTS

Mutation type	Baseline code element	Mutated code element	Notes
VALUE	(x==N)	(x==M)	N≠M
AND	condition1 AND condition2	condition1 OR condition2	Alternatively, OR changed for AND
NOT	condition	NOT(condition)	Alternatively remove NOT

To detect the defects caused by input mutations in the particular SUT instance, we created Selenium WebDriver[2] automated tests. The test scenario consisted of the following steps: (1) login to the system, (2) select the project, (3) go to the issue entry form, (4) enter the issue by prepared testing data, (5) submit the issue, (6) test the verification mechanism for the allowed issue data, (7) display the saved issue to test if it was saved correctly. The sequences (3)-(7) repeated for particular combinations of the test data, which were parametrized in a data grid structure. The experimental setup is outlined in the Fig. 2.

The test data (which were entered to the SUT during the experiments) were generated by our strategy PSTG that is developed recently. PSTG is a CIT strategy that uses Particle Swarm Optimization (PSO) to generate the combinatorial interaction test suite. Details about the PSTG test generation tool can be found in [34], [35]. The strategy generates the test suites by entering the specification of input models. We prepared different sets of test data: starting from $2-way$, where the test data combinations generated with full all pairs criterion for the input values and $t-way$, where the test data combinations were generated with full $t-way$ criterion.

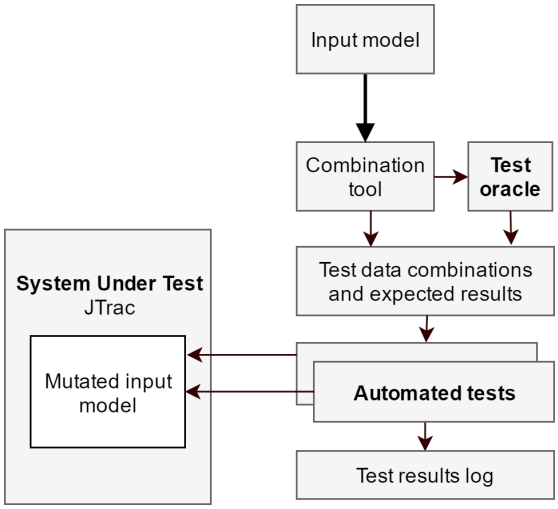

Fig. 2. Experiment setup

The following section shows and discusses the results of the evaluation of these test suites.

[2]http://www.seleniumhq.org/projects/webdriver/

V. Results and Discussion

Using our PSTG tool, we have generated $t-way$ test suites starting from $2-way$ to $6-way$. We have started by 48 different test cases in the $2-way$ test suite and 322 different test cases in the $3-way$ test suite. We used baseline SUT (without inserted faults) to create a test oracle, determining the correct expected results of individual test cases. These expected results were added to the input data grid of the automated tests exercising particular mutated SUTs. We run each of the prepared test cases (TCs) in $2-way$ and $3-way$ test suites for all 10 mutated SUTs instances. The results are presented in Table IV.

In Table IV, DN denotes number of test cases that have detected the defects caused by the mutated input lines; EFF denotes percentage of test cases that have detected the defects caused by the mutated input lines. For three mutated SUTs (particularly 3,4 and 5), the defects were detected only when using a $3-way$ test cases. The input mutants which were not detected by $2-way$ test suite, but where detected by the $3-way$ test suite are specified in the Table V.

It is also important to know the effectiveness of each test case in addition to the test suites. These individual effectiveness results are presented in Fig. 3 for the $2-way$ test suite and Fig. 4 for the $3-way$ test suite. In Fig. 3, the graph displays the number of mutated SUTs, in which the individual test case of the $2-way$ test suite discovered a defect caused by a input mutation. By analogy, Fig. 4 displays this statistics for the $3-way$ test suite.

Fig. 5 presents the ratio of test cases detecting the input mutation defects for the $2-way$ test suite. On the $x-axis$, the number of mutated input lines in particular mutated SUTs is displayed.

Then, Fig. 6 presents the ratio of test cases detecting the input defects for the $3-way$ test suite.

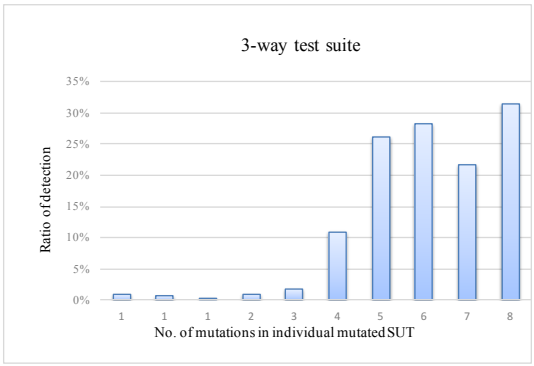

Fig. 6. Ratio of test cases detecting the mutated input defects for the $3-way$ test suite

It is worthy to mention that the other generated test suites (i.e., $4-way$ to $6-way$) may also detect faults in the program. However, for this particular application, we can see from the results that $3-way$ test suites can detect all the mutants successfully.

In addition to the effectiveness of the test suites as a whole, another important observation can be drawn from the results, which is the effectiveness of each test case. As can be seen from Fig.5 and Fig. 6, the mutant detection ration is not distributed uniformly among the test cases. Each test case has a different capability of detection. This could be beneficial in term of test case prioritization while accompanied by regression testing. Here, we can rearrange the test cases in the test suite in which those test cases with higher detection ratio can be prioritized when running in the future.

In particular, 19 out of 48 test cases of the $2-way$ test suite have not detected any defect in all of 10 mutated SUTs. This means, 39,6% of $2-way$ test suite test cases have not detected any defect in all of 10 mutated SUTs used in the experiment. One test case of the $2-way$ test suite has detected 1,176 defects in all of SUTs in average. For $3-way$ test suite, 130 out of 322 test cases have not detected any defect in all of the mutated SUTs, which is 40,4% in ratio. One test case of the $3-way$ test suite has detected 1,230 defects in all of 10 SUTs in average.

When we consider these statistics as a certain measure of test set efficiency, for 2-way and 3-way test suites this efficiency is is practically comparable. From an overall economic point of view, 2-way test set seems more efficient, taking into account the total number of test cases in $2-way$ and $3-way$ test sets. Nevertheless, the significant difference is more capability of $3-way$ test set to detect all the inserted defects; in our example 3 of the inserted defects were detected by $3-way$ test set only.

VI. Threats to Validity

Like any other experimental and evaluation research, this research has faced few threats to validity. We have tried to eliminate the effect of those threats. However, some threats are out of our control in research. The first threat is the generalization and expansion of the results. For this experimental object of this research, we have selected limited input elements for mutation. There could be different elements of the input that may take various types of mutant. In fact, this threat depends on the aim of the experiment. Here our aim is not to evaluate JTrac extensively, but we are using it to proof the effectiveness.

The second possible threat is the use of front-end testing and model-based mutation testing to detect the defective behavior of the SUT. Although it can be thought as a threat, we took this approach to simulate real-life test cases.

Another threat is that the set of used mutated input types are limited. Here, we tried to estimate the most likely developer's mistakes. It is hard to predict which defects are made by the developers in the input line for the particular case. Quality and consistency of the design documentation, seniority of the development staff and coding standards have an impact on the particular case.

The use of one generation could form a threat to validity. In the literature, there are different approaches for constructing combinatorial interaction test suites. Each approach may lead to different effectiveness impact on the SUT. Here, in line with

TABLE IV
TEST RESULTS

Mutated SUT ID	DN for $2-way$	EFF for $2-way$	DN for $3-way$	EFF for $3-way$
1	1	2,08%	3	0,93%
2	1	2,08%	2	0,62%
3	0	0,00%	1	0,31%
4	0	0,00%	3	0,93%
5	0	0,00%	6	1,86%
6	3	6,25%	35	10,87%
7	12	25,00%	84	26,09%
8	12	25,00%	91	28,26%
9	10	20,83%	70	21,74%
10	17	35,42%	101	31,37%

TABLE V
INPUT MUTANTS WHICH WERE NOT DETECTED BY $2-way$ TEST SUITE

Mutated SUT ID	Mutation ID	Baseline input element	Mutated input element
3	1	IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid	IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid
4	1	IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid	IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid
4	2	IF ((issue_severity==2) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid	IF ((issue_severity==3) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid
5	1	IF ((issue_severity==1) AND ((issue_priority==4) OR (issue_priority==5)) THEN the combination is not valid	IF ((issue_severity==1) AND ((issue_priority==5) OR (issue_priority==5)) THEN the combination is not valid
5	2	IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid	IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid
5	3	IF ((issue_severity==2) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid	IF ((issue_severity==3) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid

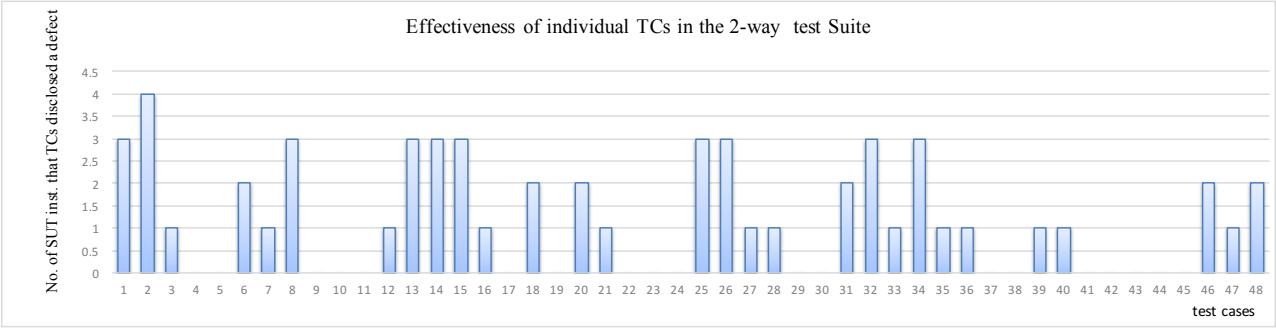

Fig. 3. Effectiveness of the individual test cases for the $2-way$ test suite

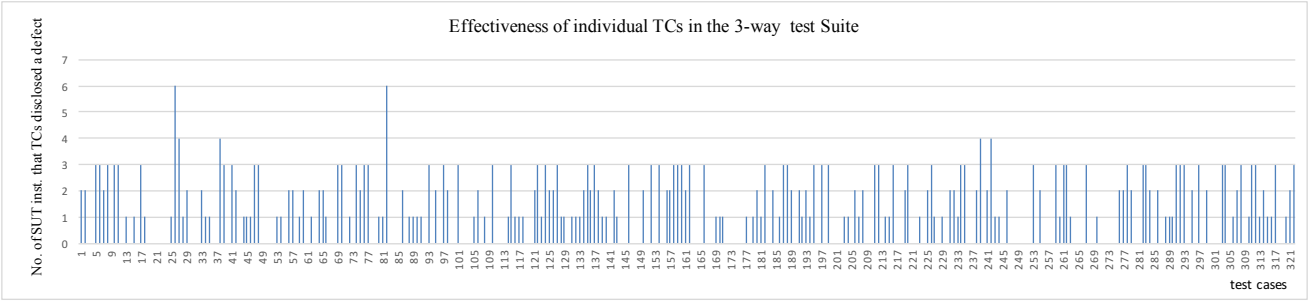

Fig. 4. Effectiveness of the individual test cases for the $3-way$ test suite

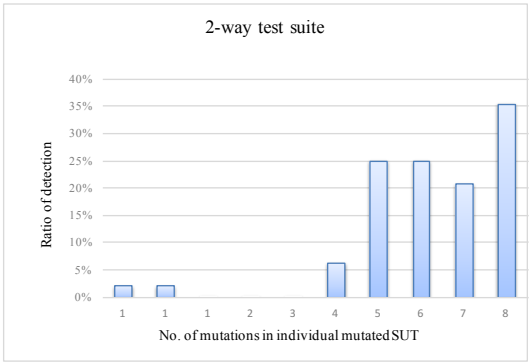

Fig. 5. Ratio of test cases detecting the mutated input defects for the $2-way$ test suite

our aim, we are not showing the effectiveness of each CIT strategy. However, we keen to demonstrate the effectiveness of CIT within the model-based mutation testing for a new case study. Here, in contrast to the traditional code-based mutation testing, we can detect different input mutants by following the model-based mutation testing process.

VII. Conclusion

In this paper, we have investigated a new case study of CIT for model-based mutation testing. We have examined different combinatorial interaction test suites on a well-known established open source software. The experiments aimed to find input model mutants that we injected into the SUT to check the effectiveness of the test suites. In addition to the critical application of CIT that can be seen in the paper, we have also shown how to use CIT in the context of model-based mutation testing as a variant of traditional code-based mutation testing. We have also shown that pairwise testing is not sufficient to detect all the input mutants for this case study. The results revealed that $3-way$ test suite is effective also to detect some of those not detected faults by pairwise test suites. This research is an active research project currently, and we are trying to investigate more case studies with more mutated inputs for CIT and its effectiveness in the future. As part of our future research, we are also planning to examine the effectiveness of different CIT strategies to study the effect of test construction and parameter arrangements on the effectiveness of the whole test suites.

References

[1] R. N. Kacker, D. Richard Kuhn, Y. Lei, and J. F. Lawrence, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement*, vol. 46, no. 9, pp. 3745–3752, 2013.

[2] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Information and Software Technology*, vol. 66, no. 0, pp. 13–29, 2015.

[3] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Softw*, vol. 13, 1996.

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

[5] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: a general strategy for t-way software testing," in *4th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pp. 549–556, IEEE Computer Society. 1253335 549-556.

[6] A. Hervieu, D. Marijan, A. Gotlieb, and B. Baudry, "Practical minimization of pairwise-covering test configurations using constraint programming," *Information and Software Technology*, vol. 71, pp. 129–146, 2016.

[7] K. Z. Zamli, M. F. J. Klaib, M. I. Younis, N. A. M. Isa, and R. Abdullah, "Design and implementation of a t-way test data generation strategy with automated execution tool support," *Information Sciences*, vol. 181, no. 9, pp. 1741–1758, 2011.

[8] X. Yuan, M. B. Cohen, and A. M. Memon, "Gui interaction testing: incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[9] C. S. Cheng, "Orthogonal arrays with variable numbers of symbols," *The Annals of Statistics*, vol. 8, no. 2, pp. 447–453, 1980.

[10] C. Colbourn, G. Kéri, P. R. Soriano, and J.-C. Schlage-Puchta, "Covering and radius-covering arrays: Constructions and classification," *Discrete Applied Mathematics*, vol. 158, no. 11, pp. 1158–1180, 2010.

[11] A. Hartman, *Software and Hardware Testing Using Combinatorial Covering Suites*, vol. 34 of *Graph Theory, Combinatorics and Algorithms*. Springer US, 2005.

[12] J. N. Cawse, *Experimental design for combinatorial and high throughput materials development*. Wiley-Interscience, 2003.

[13] D. E. Shasha, A. Y. Kouranov, L. V. Lejay, M. F. Chou, and G. M. Coruzzi, "Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era," *Plant Physiology*, vol. 127, no. 4, pp. 1590–1594, 2001.

[14] D. S. Hoskins, C. J. Colbourn, and D. C. Montgomery, "Software performance testing using covering arrays: Efficient screening designs with categorical factors," in *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, (New York, NY, USA), pp. 131–136, ACM, 2005.

[15] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[16] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," *Journal of Combinatorial Optimization*, vol. 15, no. 1, pp. 17–48, 2008.

[17] G. Fraser and A. Gargantini, "Generating minimal fault detecting test suites for boolean expressions," in *Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 37–45, 2010.

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

[19] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 26–36, ACM, 2013.

[20] P. A. da Mota Silveira Neto, I. d. Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[21] D. R. Sulaiman and B. S. Ahmed, "Using the combinatorial optimization approach for dvs in high performance processors," in *International Conference on Technological Advances in Electrical Electronics and Computer Engineering (TAEECE)*, pp. 105–109, 2013.

[22] M. A. Sahib, B. S. Ahmed, and M. Y. Potrus, "Application of combinatorial interaction design for dc servomotor pid controller tuning," *Journal of Control Science and Engineering*, vol. 2014, p. 7, 2014.

[23] B. S. Ahmed, M. A. Sahib, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Optimum design of PI$^{\lambda}$ D$^{\mu}$ controller for an automatic voltage regulator system using combinatorial test design," *PLOS ONE*, vol. 11, pp. 1–20, 11 2016.

[24] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "Application of particle swarm optimization to uniform and variable strength covering array construction," *Applied Soft Computing*, vol. 12, no. 4, p. 1330â1347, 2012.

[25] B. S. Ahmed and K. Z. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2171–2185, 2011.

[26] B. S. Ahmed, M. A. Sahib, and M. Y. Potrus, "Generating combinatorial test cases using simplified swarm optimization (sso) algorithm for automated gui functional testing," *Engineering Science and Technology, an International Journal*, vol. 17, no. 4, pp. 218–226, 2014.

[27] R. C. Bryce and C. J. Colbourn, "Test prioritization for pairwise interaction coverage," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST '05, (New York, NY, USA), pp. 1–7, ACM, 2005.

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

[29] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 45–54, 2004. 1007519.

[30] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098 – 1107, 2011. Special Section on Mutation Testing.

[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 649–678, Sept. 2011.

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

[33] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing – approach and case studies," *Science of Computer Programming*, vol. 120, pp. 25 – 48, 2016.

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

[35] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "Constructing a t-way interaction test suite using the particle swarm optimization approach," *International Journal of Innovative Computing, Information and Control (IJICIC)*, vol. 8, no. 1, pp. 431–452, 2012.