Code-Aware Combinatorial Interaction Testing

Bestoun S. Ahmed, Angelo Gargantini, Kamal Z. Zamli, Cemal Yilmaz, and Miroslav Bures

***Abstract*—Combinatorial interaction testing (CIT) is a useful testing technique to address the interaction of input parameters in software systems. In many applications, the technique has been used as a systematic sampling technique to sample the enormous possibilities of test cases. In the last decade, most of the research activities focused on the generation of CIT test suites as it is a computationally complex problem. Although promising, less effort has been paid for the application of CIT. In general, to apply the CIT, practitioners must identify the input parameters for the Software-under-test (SUT), feed these parameters to the CIT tool to generate the test suite, and then run those tests on the application with some pass and fail criteria for verification. Using this approach, CIT is used as a black-box testing technique without knowing the effect of the internal code. Although useful, practically, not all the parameters having the same impact on the SUT. This paper introduces a different approach to use the CIT as a gray-box testing technique by considering the internal code structure of the SUT to know the impact of each input parameter and thus use this impact in the test generation stage. We applied our approach to three reliable case studies. The results showed that this approach would help to detect new faults as compared to the equal impact parameter approach.**

***Index Terms*—Software testing , Combinatorial interaction testing (CIT) , Mutation testing, Fault finding, Program analysis Code coverage analysis.**

I. Introduction

Modern software systems are increasingly getting large in terms of size, functionality, input parameters, and configurations. The different interaction (combinations) of input parameters or configurations may cause faults while running the system. Kuhn *et al.* [1] reported many interaction faults in mission-critical and other software systems. Combinatorial interaction testing (CIT) (sometimes called $t - way$ testing, where t is the interaction strength) offers a sampling strategy that can effectively and efficiently sift out only fewer interactions to equate the otherwise impossible exhaustive testing of all interactions. To generate the CIT test suite, covering arrays were used to sample the inputs of the SUT based on the input interaction coverage criteria. This approach has been used for input combination and configuration testing as a black-box approach. To sample the input parameters in the combinatorial input testing, each row in the covering array represents a complete set of input to the software-under-test (SUT), while each row in configuration testing represents a configuration setting of the SUT [2]. In both cases, the testing process is done as black-box testing where the internal code is not used during the test generation.

While the black-box approach with the CIT is useful in many applications, not all the test cases are equally effective in finding faults especially in the input parameter CIT testing [3]. The current test case generation algorithms do not consider the parameter impact on the SUT. In fact, CIT is generally used as a black-box testing strategy. Practically, not all the input parameters have the same impact on the internal code structure of the SUT. One way of defining the effect of a parameter is to consider those parameters that cover more lines of code to have more impact. Considering those parameters to generate or refactor the generated test cases may lead to a more practical CIT.

In this paper, we argue that CIT could be more useful when it also considers the internal code of the SUT. Here, we propose to mix black-box testing with white box testing (i.e., gray-box testing approach) where we extend CIT by reusing the information that is coming from the code. We introduce our code-aware approach to generate combinatorial interaction test suites. The approach relies on the generation of more effective test suites by considering the internal code structure of the SUT through the feedback from the code coverage analysis. Here, the test generation process relies on a preassessment and analysis of the internal code structure of the SUT to know the sensitivity for each input parameter and thus knowing the impact of each one of them. By understanding this impact, then, instead of using uniform interaction strength among the input parameters, we put more focus on those impacted parameters by considering higher or even full interaction strength. Hence, we use mixed strength instead of uniform strength. Our aim is to generate more effective test cases (in term of better fault detection capability) by identifying those input parameters which are practically affect the SUT.

The rest of this paper is organized as follows. Section II gives background information on the CIT and essential concepts of the testing and generation algorithms. Section III illustrates our method in this paper including the analysis and testing procedures. Section IV illustrates our empirical investigation of the code-aware combinatorial interaction testing approach, including the results and discussion. Section V discusses possible threats to validity. Finally, Section VI gives the concluding remarks of the paper.

II. Background and Motivation

Theoretically, the combinatorial test suite depends on a well-known mathematical object called Covering Array (CA). To represent a test suite, each row in the CA presents a test case and each column represents an input parameter of the SUT.

B. Ahmed and M. Bures are with the Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, Karlovo nam. 13, Prague, Czech Republic, email: albeybes@fel.cvut.cz

A. Gargantini is with the Department of Management, Information and Production Engineering, University of Bergamo, Italy, email: angelo.gargantini@unibg.it

C. Yilmaz is with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey, email: cyilmaz@sabanciuniv.edu

K. Zamli is with the Faculty of Computer Systems and Software Engineering, University Malaysia Pahang, Gambang, Malaysia, email: kamalz@ump.edu.my

Formally, a $CA_\lambda(N; t, k, v)$ is an $N \times k$ array over $(0, ..., v-1)$ such that every $B = \{b_0, ..., b_{t-1}\}$ is λ-covered and 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 $B = \{b_0, ..., b_{t-1}\} \supseteq \{0, ..., k-1\}$ [4], [5]. In this case, each tuple is to appear at least once in a CA.

In the case when the number of component values varies, this can be handled by Mixed Covering Array (MCA). A $MCA(N; t, k, (v_1, v_2, ..., v_k))$, is an $N \times k$ array on v values, where the rows of each $N \times t$ sub-array covered and all t interactions of the values from the t columns occur at least once. For more flexibility in the notation, the array can be presented by $MCA(N; t, v_1^{k_1} v_2^{k_2} .. v_k^k)$.

In real-world complex systems, the interaction strength may vary between the input parameters. In fact, the interaction of some input parameters may be stronger than other parameters. Variable strength covering array (VSCA) is introduced to cater for this issue. A $VSCA(N; t; k, v, (CA_1, ..., CA_k))$ represents $N \times p$ MCA of strength t containing vectors of CA_1 to CA_k, and a subset of the k columns each of strength $> t$ [6]. Also, practically, not all the inputs are interacting and having an impact on each other. Some parameters may not interact at all. Here, it is not necessary to cover all the interactions of the parameter. Presenting those parameters even in one test case would be enough.

CIT used these mathematical objects as a base for the testing strategy of different applications. A wide range of applications appeared in the literature. Mainly, CIT used in software testing and program verification. There are many applications in this direction, for example, fault detection, and characterization [7], graphical user interface testing (GUI) [8], model-based testing and mutation testing [9]. There are many more applications of CIT in software testing. Comprehensive surveys about these applications can be found in [10]–[12]. The concepts of CIT also finds its way to other fields rather than software testing. For example, it has been used in the satellite communication testing, hardware testing [13], advance material testing [14], dynamic voltage scaling (DVS) optimization [15], tunning the parameter of fractional order PID controller [16], and gene expression regulation [17].

In most of the applications, the combinatorial interaction test suite is generated by establishing a coverage criterion. Here, the coverage criterion is to cover the $t - tuples$ of the input parameter at least once to generate the CA. A few research in the literature considered some other input attributes during the generation in addition to the $t - tuple$ coverage criteria. For example, Yilmaz [2] considered the test case-specific interaction constraints which are test cases related to the configurations. Demiroz and Yilmaz [18] also introduced the cost-aware covering arrays that generate the test cases based on a given cost function by modeling the actual cost of testing in addition to the standard $t - tuple$ coverage criteria.

Almost, in all applications, the SUT is considered as a black-box system by the generation tool. However, in practice, not all the parameters have the same impact on the internal code of the SUT. Logically, each value of the input parameters may have a different impact on the program. Taking into account the code coverage as an example, those parameters' tuples with higher code coverage may have a higher chance to cause faults. In this paper, we have considered this situation to take the CIT in a gray-box testing approach by analyzing the program internals.

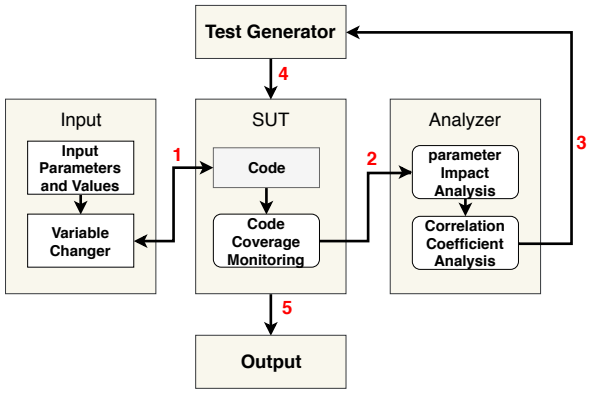

Fig. 1. Experimental method

III. Method

For this study, we only consider those software systems which can be tested using the CIT approach. Here, the test suite can be represented as a CA with more than two input parameters. Each parameter has different values. We hypothesize that if we measure the parameter strength based on code coverage, we can figure out how much that input parameter has impact on the SUT. Larger impact means that the parameter covers more lines of SUT code and thus it is more prone to failures. Hypothetically, those parameters may have better fault detection rates. Let X and Y be two parameters of a program P and let their code coverage $C_X > C_Y$.

To analyze the parameter impact of each SUT, we have followed several systematic experimental steps. Figure 1 shows these steps.

First (step 1 in Fig. 1) we have identified the parameters and the values and we have called the code with a set of tests. To avoid the omission of parameters' values effect on each other, we have tried two different possibilities and combinations of the values. First, we attempted to measure the coverage while we varied the values of a specific parameter and made the rest constant. Second, we also tried different combinations of the values. Both approaches lead to the same conclusion as we are using the deviation at the end. Measuring the code coverage when we vary the value of a specific parameter (by the Variable Changer in Figure 1) while we make the rest constant will lead to conclude the impact of that parameter. However, in some situation, the code coverage may be affected by some other values of the other parameters. In fact, we considered this situation also to measure the code coverage of two parameters (i.e., pairwise) while making the others constant. Hence, for fair experimental results, we considered all the possibilities of the parameter effects on each other.

By measuring the code coverage (step 2 in Fig. 1), we are able to calculate each parameter impact by measuring the deviation of code coverage when entering different parameter values. For a fair experimental procedure, we used the best

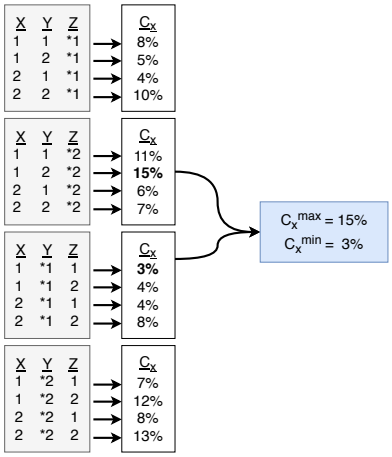

Fig. 2. Code coverage measurement method

and worst code coverage situation as maximum and minimum code coverage.

To measure the code coverage, we used an automated scripting framework to monitor and measure the impact of each input parameter. To calculate the impact of a parameter, we calculate the code coverage deviation as in Eq 1.

$$I_p = C_p^{max} - C_p^{min} \tag{1}$$

where I_p is the parameter impact, C_p^{max} is the maximum code coverage of parameter p, and C_p^{min} is the minimum code coverage of parameter p. The Eq 1 can be used to calculate the impact of pairwise parameters also, which can be used for the correlation coefficient later in Eq 2. To better illustrate the code coverage analysis process, Figure 2 shows a simplified example for a system with three input parameters (X, Y, and Z) where each parameter has two values (1 and 2). For illustration, we assign random numbers to the code coverage for each test case. The aim here is to measure the impact of parameter X. Here, the maximum and minimum code coverage of X are 15% and 3% respectively. In fact, parts of these experimental records can be used mutually for the impact analysis of Y and Z also. Here, we consider the pairwise possibilities for the parameter values to consider all the parameters between X and Y, while assigning a constant values to Z (values with "*" sign). To avoid the omission of other values, we also change the constant values for the parameters for each pairwise experiment.

Using this approach to know the impact of the parameters, we can calculate the interaction weight by calculating the correlation of the parameters. For parameters X and Y, the correlation would be as in Eq.2:

$$Corr(X, Y) = \frac{I_{XY}}{I_X * I_Y} \tag{2}$$

where I_X and I_Y are the impact of the parameters X and Y respectively, and the I_{XY} is the impact of both of them. As mentioned previously, we have undertaken a careful code coverage monitoring to isolate the effects.

TABLE I
CONCEPTUAL MODEL OF PARAMETER CORRELATION MAPPING

parameter	P_1	P_2	...	P_n
P_1	-	$Corr(P_1, P_2)$	...	$Corr(P_1, P_n)$
P_2	$Corr(P_2, P_1)$	-	...	$Corr(P_2, P_n)$
...	...	...	-	...
P_n	$Corr(P_n, P_1)$	$Corr(P_n, P_2)$	...	-

Now, with this correlation, for each tuple of parameters, we create an $n \times n$ square matrix, where n is the number of parameters, as in Table I.

Using the data in Table I, we can select and device the higher impacted tuples by assigning a partner of each parameter p_i, as in Eq.3.

$$partner(p_i) = \arg\max_{p_j} Corr(p_i, p_j) \tag{3}$$

Note that since the matrix in Table I is reflexive and diagonally mirrors itself, the same tuple will be present twice. For example, if the best partner for p_1 is p_2, then the best partner for p_2 is p_1. Therefore, this tuple will appear in the matrix both as $Corr(p_2, p_1)$ and $Corr(p_1, p_2)$. Both instances represent the same tuple however.

In practice, one can assign higher interaction strength to not only pairs of parameters, but also to a set of three and more. Assigning higher interaction strength to a set of all parameters would result in no comparative change in strength impact between the individual parameter subsets. If one would decide to assign a higher strength to three or more parameters, it can be done in two ways. The first approach is to create a multi-dimensional matrix (n-dimensional for sets of n), and the methodology is the same. Alternatively, it is possible to spot a good set of three candidates in a two-dimensional matrix.

Due to the goal of this paper, we did not pay attention to the execution cost of the test suites. As mentioned previously, we aim to assess the effectiveness of our code-aware CIT approach via an experimental study. Based on this experimental study, these steps can be automated and abstracted to minimize the execution cost for the ease of use in the industry.

A. Mutant generation and fault seeding

To analyze the effectiveness of our approach, we generate different types of mutants to be injected into the subjected programs for experiments. We used $\mu java$[1], the classical java mutation tool, to generate the mutants. $\mu java$ is a mutation system for Java programs. It automatically generates mutants for both traditional mutation testing and class-level mutation testing. $\mu java$ can test individual classes and packages of multiple classes. Tests are supplied by the users as sequences of method calls to the classes under test encapsulated in methods in JUnit classes.

For the fault seeding, we have seeded 35 types of faults, as defined by the $\mu java$ documentation. Two levels of faults were used here, method-level faults and class-level faults. Mutants in the method-level fault seeding are based on changing

[1]https://cs.gmu.edu/ offutt/mujava/

TABLE II
TYPES AND ABBREVIATION OF METHOD-LEVEL FAULTS USED FOR THE EXPERIMENTS

Abbreviation	Meaning
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement
SDL	Statement DeLetion
VDL	Variable DeLetion
CDL	Constant DeLetion
ODL	Operator DeLetion

operators within methods, or the complete statement alteration. In this analysis, 16 different types are considered. Table II shows those method-level faults and the correspondence abbreviations. Mutants based on class-level fault seeding are based on changing operators within methods, or the complete statement alteration. In this analysis, 29 different types are considered. Table III shows those class-level faults and the correspondence abbreviations.

B. Test case generation

To generate the test cases, we used ACTS3.0[2], the automated CIT tool that contains many algorithms to generate the combinatorial interaction test suites. ACTS is a well-known combinatorial interaction test generation tool that supports the generation of different interaction strength ($1 \leq t \leq 6$). The tool provides both command line and GUI interfaces. The tool also offers the flexibility to address the interaction strength for different set and subsets of input parameters, (i.e., mixed strength and variable strength interaction). To avoid the randomness of the test generation algorithms and to assure fair experiments statistically, we used the deterministic test generation algorithms in ACTS.

C. Parameter impact analysis

Here, we adopted code coverage monitoring tools in our environment to analyze the code coverage during the first round of test execution. By analyzing the code coverage, we know the impact of the parameters with respect to the code. We adopted Sofya[3] Java bytecode analysis tool for the code coverage analysis. Sofya is a tool designed to provide analysis capabilities for Java programs by utilizing the Bytecode Engineering Library (BCEL) to manipulate the class files. We have also used the base JetBrains[4] IntelliJ IDE for the code coverage measurement. The first round takes several iterations depending on the input parameters and the value of each one of them. As previously mentioned, the program examine each parameter to identify C_p^{max} and C_p^{min} in Eq.1.

IV. EMPIRICAL INVESTIGATION

In this section, we illustrate our empirical investigation. Here, three software subjects were used as case studies for the investigation. We first describe these case studies and then the fault injection and code analysis procedures. During this empirical investigation, we aim to answer three main research questions (RQ)s:

- RQ1: To which extent the input parameters affect the internal code of the SUT?
- RQ2: How the input parameters are correlated?
- RQ3: How does the new approach improve the effectiveness of fault detection in the CIT?

A. Case studies

We chose three Java-based subject programs for the experiments. Two programs from Software-artifact Infrastructure Repository (SIR)[5] and one program from freesourcecode[6] software repositories. These programs are well-known experimental programs used for experimental purposes in other research studies. We choose "Replicated workers", "Groovy", and "Body calculator[7]" for the case studies.

The replicated workers is an implementation of a standardized Replicated Workers problem. In some parallel algorithms, the number of specific computing tasks is not known in advance. To control the allocation of the tasks of the replicated workers, a work pool is used. The replicated workers program has 342 lines of code. The groovy snippet is a part of the core for Apache Groovy, a multi-faceted language for the Java platform. The groovy program has 361 lines of code. The body calculator application/applet is a medical program used for fat percentage, body mass index, Basal Metabolic Rate, Ideal Weight, and the Calorie Intake. The body calculator program has 910 lines of code.

B. Experimental procedure

To follow the methodology given in Section III, we have created two sets of test cases for each case study. One reference set without parameter impact, and another set using our approach by considering the parameter impact. To know the effectiveness, we have seeded faults into the programs. We run each program with and without seeded faults to kill the mutants by recognizing the differences. Based on our approach, we then regenerate the test suites by putting interaction strength on those parameters which are correlated more to each other by considering the correlation coefficient, as illustrated in Table I. For example, if the correlation coefficient of two parameters A and B is equal to 0.9, it means they are highly correlated to each other, we put the interaction strength on both of them. On another hand, if the correlation coefficient between the two parameters C and D is equal to 0.1, it means

[2]https://bit.ly/2s2IajU
[3]http://sofya.unl.edu/
[4]https://www.jetbrains.com/
[5]http://sir.unl.edu/portal/index.php
[6]freesourcecode.net
[7]https://bit.ly/2rrUolM

TABLE III
TYPES AND ABBREVIATION OF CLASS-LEVEL FAULTS USED FOR THE EXPERIMENTS

Feature	Abbreviation	Meaning
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-specific features	JTI	This keyword insertion
	JTD	This keyword deletion
	JSI	Static modifier insertion
	JSD	Static modifier deletion
	JID	Java Member variable initialization deletion
	JDC	Java-supported default constructor deletion
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Access or method change
	EMM	Modifier method change

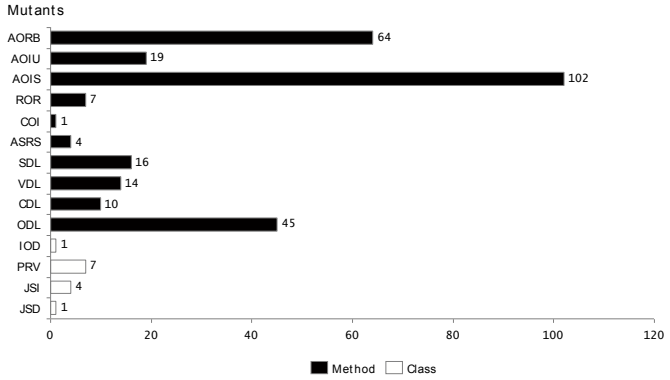

Fig. 3. Type and number of seeded faults into the Replicated Worker program

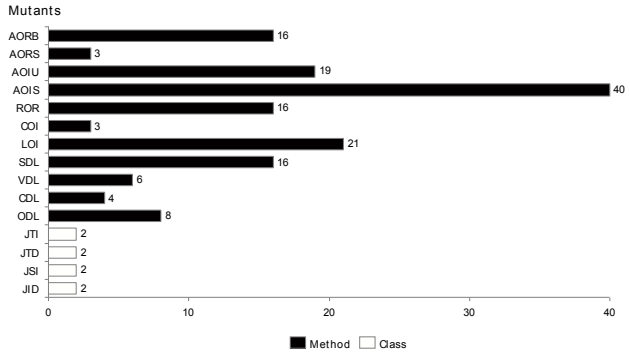

Fig. 4. Type and number of seeded faults into the Groovy program

that the correlation is too low between them, we assign a "don't care" value for both of them. Hence, they will be presented in the test suites, but we don't care about the full coverage of their values. We run the newly generated test suite to identify the differences in fault detection.

C. The seeded faults

As previously mentioned, we have seeded many faults in each subjected program for the case study. Using $\mu java$, we have seeded 295 faults into the Replicated workers program, 160 faults into the Groovy program, and 1512 faults into Body calculator program. Figures 3 - 5 shows the number and the type of all these faults for each subjected program individually.

D. Observations

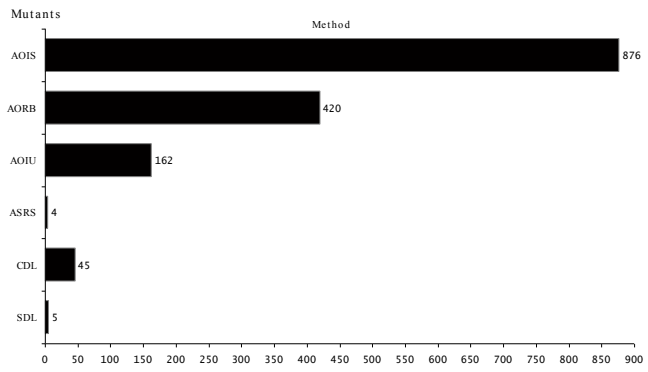

Fig. 5. Type and number of seeded faults into the Body calculator program

TABLE IV
REPLICATED WORKERS PARAMETER IMPACT ANALYSIS

Parameter Name	Type	Covered	Uncovered	Examples
num_workers	int	30%	51%	(1,2,5)
num_items	int	8%	73%	(1,2,5)
min	float	4%	77%	(1,10,15)
max	float	4%	77%	(1,10,15)
epsilon	float	81%	0%	(0.05,0.1,0.25)

1) RQ1. Parameter analysis - the effect on the internal code: Based on the method in Section III, we have analyzed each program. As illustrated in the Figure 2, we measured the code coverage of each input parameter in the testing framework. We changed the variable of the input parameter and measured the coverage to know the impact of the parameter on the internal code using Eq.1. For the case studies used in this paper, we double check the results also by manually inspection and reviewing the code of each program. Tables IV - VI show the empirical results of the parameter impact analysis using the code coverage.

Knowing the effect of each parameter on the program by trying all of its values gives a precise analysis of the sensitivity of the SUT by each one of them. Here, it is clear that not all the parameters have the same impact on the program. For example, the parameters "epsilon" and "num_workers" have more impact on the replicated workers program as compared to other input parameters as they are covering more LOC. Similarly, the parameters "threadCount" and "neck" have more impact on the Groovy and body calculator programs respectively.

It should be mentioned here that the amount of coverage by each parameter gives the number of LOCs that is related only to that parameter and excluding the related LOCs to the other parameters. However, there are still common LOCs related to two or more parameter. Hence, we don't expect to have full coverage of LOCs by only one or even two input parameters. In fact, the full coverage of LOCs is not achievable even with all input parameters. This situation is due to different reasons such as poor programming and development practice of the SUT itself.

2) RQ2. Correlation analysis - the input parameters' correlation: Using the empirical results from parameter impact analysis, we can calculate the correlation among each tuple of input parameters. We used the standard correlation calculation in statistics in Eq. 2 to know the interaction weight between the tuples as described in Section III. Tables VII - IX show the correlation among the tuples of input parameters for each case study.

Analyzing the correlation between the tuples of parameters gives a clear understanding of how each value of these parameters are related to each other and the strength of the relationship. We consider a relationship stronger as the value of the correlation approached to "1". Using this criterion for classification, we can categorize the tuples. For example, the parameters "min" and "num_items" in the replicated workers program are highly related to each other as they have correlation value equal to 1. Similarly, the parameters "threadCount" and "concurrentReads" in the Groovy program are highly related to each other. In contrast, the interaction weight (i.e., relationship) between the "num_items" and "epsilon" parameters is weak. As previously mentioned, we can put higher strength or full strength on those highly-related parameters and free the weakly-related parameters, i.e., marking them as "don't care" value in the test generation tool. Following this approach, we assure that those highly-related tuples will be fully covered in the test suite while the other tuples will appear in the test suite but not fully covered.

Table X shows the size of the generated test suite for each case study after and before the consideration of the impact analysis. It is noticeable that the size of the test suite decreased by a few test cases with the Replicated worker and the Groovy case studies. As we can see, for these two case studies there are a few parameters and not all of them are highly related to each other. As mentioned, during the test generation, we assigned "don't care" values to those parameters. As a result, the fewer (but effective) test cases were generated. Although it is not the aim of our study, this shows that in some cases our approach could also help to generate efficient test cases by generating fewer test cases.

As for the Body Calculator case study, there is a higher number of parameters than the other two case studies. As we can see from the results, those parameters are related to each other. As a result, we considered different interaction strength during the generation. Hence, the number of test cases goes higher.

3) RQ3. Assessment of fault detection - effectiveness of the new approach: As mentioned previously, we have injected different mutants into each program for possible fault detection. The experiments aim to know the effectiveness of our approach to detect new faults as compared to the classical black-box CIT.

Figures 6 - 8 show that, in general, the CIT is an effective approach to detect faults. The figures also show that using our approach (mixed strength), we can detect more faults as compared to pairwise (i.e., 2-way) testing. Here, the code-aware CIT can detect more faults by giving interaction strength to those highly correlated parameters that have been measure during the analysis process at the first stage.

As in case of the replicated worker program in Figure 6, it is clear that there is 4% higher mutation score as compared to 2-way test suite. Similarly, we can see the results of the Groovy program in Figure 7. Here, with the new approach, we can detect 16 more faults. Moreover, we can see in Figure 8 that more faults can be detected in case of Body calculator program. Here, 138 more faults were detected as compared to the 2-way test suite.

We note that there are still many faults alive and our test suites have not killed them. Practically, it is not possible to kill all the mutants with a combinatorial interaction test suite. Those faults may be killed using some other test generation algorithms as they may not be interaction faults. The alive faults may also be detected by combinatorial interaction test suites when the interaction strength is greater than two (i.e., $t > 2$). In this paper, we only aimed to show the effectiveness of the new approach. As far as the approach is effective, it can be used with higher strength easily.

TABLE V
GROOVY PARAMETER IMPACT ANALYSIS

Parameter Name	Type	Covered	Uncovered	Examples
threadCount	int	35%	8%	(0,1,2,8)
DEFAULT_INITIAL_CAPACITY	int	5%	38%	(0,1,100,1000)
DEFAULT_LOAD_FACTOR	float	5%	38%	(0,1,10,100)
MAXIMUM_CAPACITY	int	4%	39%	(0,1,100,1000,10000)
concurrentReads	long	6%	37%	(0,1,10,1000)

TABLE VI
BODY CALCULATOR PARAMETER IMPACT ANALYSIS

Parameter Name	Type	Covered	Uncovered	Examples
male	boolean	1%	32%	true,false
age	int	2%	32%	0,10,20,50
weight	int	2%	31%	0,35,60
waist	int	1%	32%	0,30,50
hips	int	2%	31%	0,15,30
neck	int	16%	17%	0,100,200,400
heightArrayNum	int	1%	32%	0,1,3,5

TABLE VII
REPLICATED WORKERS PARAMETER CORRELATION OVERVIEW

Correlation	num_workers	num_items	min	max	epsilon
num_workers	-	0.947	0.941	0.941	0.739
num_items	0.947	-	1.000	1.000	0.090
min	0.941	1.000	-	1.000	0.047
max	0.941	1.000	1.000	-	0.047
epsilon	0.739	0.090	0.047	0.047	-

TABLE VIII
GROOVY PARAMETER CORRELATION OVERVIEW

Correlation	threadCount	DEFAULT_INITIAL_ CAPACITY	DEFAULT_LOAD_ FACTOR	MAXIMUM_ CAPACITY	concurrentReads
threadCount	-	0.900	0.925	0.923	1.000
DEFAULT_INITIAL_ CAPACITY	0.900	-	0.500	0.556	0.727
DEFAULT_LOAD_ FACTOR	0.925	0.500	-	0.556	0.727
MAXIMUM_ CAPACITY	0.923	0.556	0.556	-	0.600
concurrentReads	1.000	0.727	0.727	0.600	-

TABLE IX
BODY CALCULATOR CORRELATION OVERVIEW

Parameter	male	age	weight	waist	hips	neck	heightArrayNum
male	-	0.40	0.37	0.45	0.33	0.53	0.50
age	0.40	-	0.94	0.37	0.29	0.51	0.80
weight	0.37	0.94	-	0.34	0.27	0.51	0.74
waist	0.45	0.37	0.34	-	0.97	0.58	0.45
hips	0.33	0.29	0.27	0.97	-	0.50	0.33
neck	0.53	0.51	0.51	0.58	0.50	-	0.53
heightArrayNum	0.50	0.80	0.74	0.45	0.33	0.53	-

TABLE X
THE SIZE OF THE GENERATED TEST SUITES BEFORE AND AFTER PARAMETER IMPACT CONSIDERATION

SUT	2-way	mixed
Replicated workers	28	25
Groovy	24	17
Body calculator	19	62

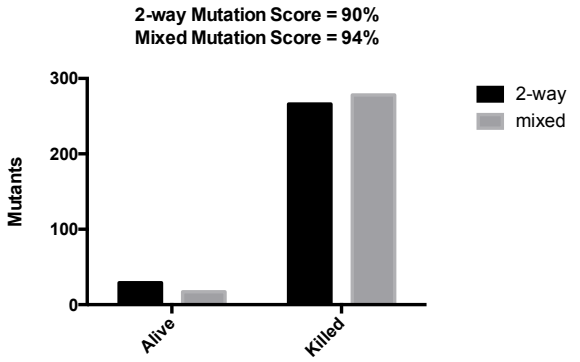

Fig. 6. Comparison of the test suite effectiveness for detecting faults in case of Replicated worker program

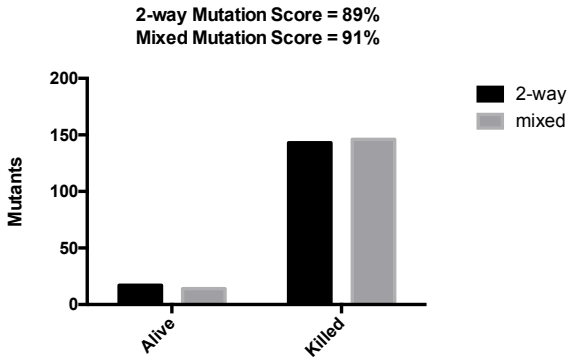

Fig. 7. Comparison of the test suite effectiveness for detecting faults in case of Groovy program

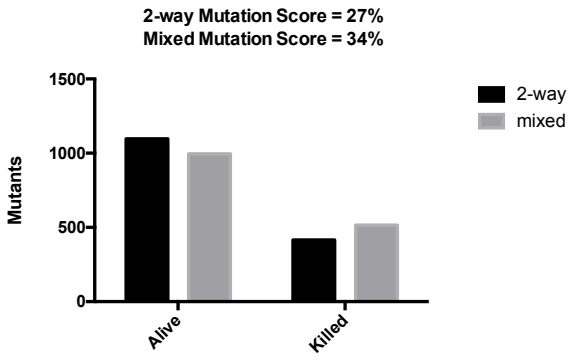

Fig. 8. Comparison of the test suite effectiveness for detecting faults in case of Body calculator program

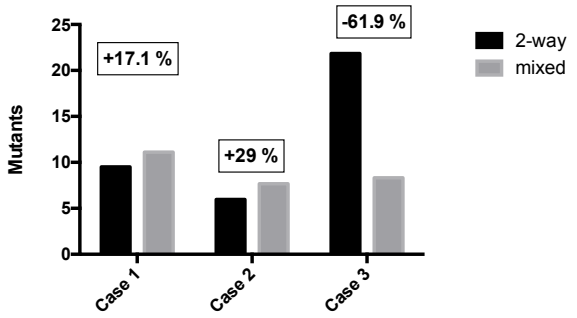

Fig. 9. Overall Efficiency of all the three case studies

If we combine the results from all three case studies, we get an impressive discussion. As can be seen in Figure 9, the code-aware CIT performed better. On average it is better by 3% in mutant detection and by 25% in efficiency (Calculated by detected mutations divided by the number of test cases). This is however largely variable based on the nature of the program. In all three cases, the code-aware CIT managed to find new mutants compared to the reference pairwise test suite. The specific number is tied to the total lines of code/mutants used, but in the first and second case studies, there is an efficiency gain of around 17% and 29%, while there is no gain in the last case study. However, as shown previously, there are still several new killed mutants that makes our approach effective.

V. THREATS TO VALIDITY

As in other empirical studies, our study is subjected to validity threats. We have tried to eliminate these threats during our experiments. We have redesigned the experiments several times to avoid different threats to validity. However, for the sake of reliability, we outline some a few significant threats that we have faced during our experiments.

Regarding the generalization of our results (i.e., external validity), we have studies only three case studies written in Java, and different results may be reported for other programs. In fact, we used more than one program to validate our approach. Also, we tried to avoid hand-generated seeded faults by using standard Java mutation tool to assure the reliability. The faults injected by the $\mu java$ are more realistic faults than the hand-generated faults.

Regarding the effect of other internal factors on our results (i.e., internal validity), there might be other factors responsible for these results that we obtain due to the instrumentation of the case studies' code. However, we have tried to run the program with and without the instrumentation and also we have double checking our experiments and manually reviewing the codes and the obtained results for a few cases. In addition, as we illustrated in Section IV-D1, internally, the parameters may affect each other, and there could be a threat that the effects of parameters were overlapping. We have tried to eliminate this threat by changing the values of the parameters systematically one by. Hence, we assure a fair measure even if it is not presenting the actual effect of that parameter on the code.

VI. Conclusion

In this paper, we have presented our new code-aware approach for conducting the combinatorial interaction testing. We showed the results of an empirical study to examine the effectiveness of our approach through three case studies. We first examined the SUT by using a code coverage framework to analyze the impact of each input parameter. Then, we used the correlation coefficient to assess the relationship of the input parameters to each other. Using these assessment and analysis steps, we were able to reconsider the generated test cases by taking those correlated parameters into account. We aim to study this approach experimentally. Although we have automated many steps during our experiments, developing an automated tool to conduct this approach is beyond the scope of this paper. In fact, this paper could serve as a strong base for a future research direction to develop an automated testing generation tool for the code-aware CIT.

As we can see from the experiments, using the parameter impact analysis can be utilized to generate effective test suites for fault detection. We named this testing process code-aware CIT. Although the fault detection rates and mutation score may vary from a program to another, the results showed that this approach is worth pursuing as another variant of CIT for practical aspects.

References

[1] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," *Computer*, vol. 42, no. 8, pp. 94–96, Aug 2009.

[2] C. Yilmaz, "Test case-aware combinatorial interaction testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 684–706, May 2013.

[3] B. S. Ahmed, "Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing," *Engineering Science and Technology, an International Journal*, vol. 19, no. 2, pp. 737 – 753, 2016.

[4] 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 Technolgy*, vol. 66, no. C, pp. 13–29, Oct. 2015.

[5] A. Hartman, *Software and Hardware Testing Using Combinatorial Covering Suites*. Boston, MA: Springer US, 2005, pp. 237–266.

[6] B. S. Ahmed and K. Z. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2171–2185, Dec. 2011.

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

[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, Jul. 2011.

[9] M. Bures and B. S. Ahmed, "On the effectiveness of combinatorial interaction testing: A case study," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017, pp. 69–76.

[10] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, 1st ed. Chapman & Hall/CRC, 2013.

[11] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, "Constrained interaction testing: A systematic literature study," *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.

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

[13] S. Y. Borodai and I. S. Grunskii, "Recursive generation of locally complete tests", journal="cybernetics and systems analysis," vol. 28, no. 4, pp. 504–508, Jul 1992.

[14] U. S. Schubert, "Experimental design for combinatorial and high throughput materials development. edited by james n. cawse." *Angewandte Chemie International Edition*, vol. 43, no. 32, 2004.

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

[16] B. S. Ahmed, M. A. Sahib, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Optimum design of pid controller for an automatic voltage regulator system using combinatorial test design," *PLOS ONE*, vol. 11, no. 11, pp. 1–20, 11 2016. [Online]. Available: https://doi.org/10.1371/journal.pone.0166150

[17] 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.

[18] G. Demiroz and C. Yilmaz, "Using simulated annealing for computing cost-aware covering arrays," *Appl. Soft Comput.*, vol. 49, no. C, pp. 1129–1144, Dec. 2016.