Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing

Mike Papadakis, Christopher Henard, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg, Luxembourg, Luxembourg
michail.papadakis@uni.lu, christopher.henard@uni.lu, yves.letraon@uni.lu

***Abstract*—Modern systems tend to be highly configurable. Testing such systems requires selecting test cases from a large input space. Thus, there is a need to systematically sample program inputs in order to reduce the testing effort. In such cases, testing the interactions between program parameters has been identified as an effective way to deal with this problem. In these lines, Combinatorial Interaction Testing (CIT) models the program input interactions and uses this model to select test cases. Going a step further, we apply mutation analysis on the CIT input model to select program test cases. Mutation operates by injecting defects to the program input model and measures the number of defects found by the selected test cases. Experiments performed on four real programs show that measuring the number of model-based defects gives a stronger correlation to code-level faults than measuring the number of the exercised interactions. Therefore, the proposed mutation analysis approach forms a valid and more effective alternative to CIT.**

***Keywords*-Mutation Analysis, Combinatorial Interaction Testing, Fault Detection**

I. Introduction

Modern software systems tend to be highly configurable and thus, they involve a vast number of parameters and configurations. This characteristic of the systems results in forming enormous input configuration spaces. We call these spaces as *input* or *test spaces* and we use the terms program inputs and test cases indifferently. Testing these systems requires exercising the programs behavior with different configurations. As a consequence, exhaustive testing is practically intractable. Since testing is a complex and costly process, a systematic and rigorous approach should be performed. To this end, the test selection should aim at choosing the tests that are potentially the best ones, i.e., revealing the highest number of defects. The question that it is raised is how to perform such a selection?

Testing the interactions between program parameters has been proposed as a possible method to answer this question [1], [2]. This proposition is based on the observation that most of the faults are triggered by the interactions between a small number of variables [3]. For instance, Kuhn *et al.* [3] have shown that interactions between two variables are able to disclose 80% of the bugs. Thus, program inputs can be systematically sampled from the test space with the aim of minimizing the number of the selected tests that exercise a specific number of interactions. In other words, this method eliminates the redundancy between the selected tests. The redundancy is measured in terms of interactions.

Testing the combinations of interactions between the input parameters has been shown to be a powerful technique [4]. This technique is called *Combinatorial Interaction Testing* (CIT) or t-wise testing and aims at selecting those tests that exercise all the interactions between any t parameters. Different values of t yield different selection criteria. Generally, a higher t value implies a stronger criterion. Also, selecting tests according to t-wise interactions subsumes, i.e., satisfies the requirements of $t-1$. Applying CIT requires a model that represents the test space. This model is composed of the input parameters associated with the possible values that they can take and some constraints. The constraints, named here as *input constraints* represent requirements on the use of variables or on the use of variable values. These constraints are very important in order to effectively perform the testing process [4], [5]. Thus, they must be fulfilled by the test cases. Test cases violating these constraints are simple false positives [4].

Generally, t-wise testing serves as a yardstick towards assessing the ability of the selected test cases to reveal faults. In other words, t-wise forms a measurement of the test suite effectiveness. It is calculated by counting the number of valid combinations that are exercised by the test suite. To this end, the ratio of the exercised t-wise combinations to the total number of the existing ones is referred to as the *interaction coverage* [4]. Therefore, given the input model and two sets of test cases, interaction coverage identifies which set is the most effective one, i.e., the one with the higher coverage. Using this information, testers can prioritize their test suites by using first the tests that cover the most interactions [4], [6], they can reduce the size of a test suite [1], [2] by removing test cases that do not cover additional interactions, and they can guide the generation or selection of new test cases based on the uncovered interactions [6], [7].

In the same lines, the present paper applies mutation analysis to the CIT input model. Thus, it introduces an alternative but more representative measure of the effectiveness of a test suite. Traditionally, mutation analysis is applied at the program code and aims at evaluating the quality of a test suite [8]. It operates by introducing artificial defects, called *mutants*, in the code of the tested program. Thus, multiple versions of the program under test are produced. Each version contains a defect that is introduced by making a slight modification. The test suites are then evaluated based on their ability to distinguish the introduced problems [8]. Contrary to the traditional approach, we apply mutation on the input model. Hence, we do not need to execute the system. We only need to evaluate whether the selected test cases satisfy or violate the altered input models. The testing process can then be performed based on the selected test cases.

In this paper, we introduce defects on the model of the program inputs. Thus, we create various input models, each one containing one defect. We apply this approach in the same way as t-wise testing is applied. However, instead of measuring the number of covered interactions, we measure the number of mutated input models that are violated by the selected tests. Therefore, we have test cases that satisfy all the constraints of the original input model but which violate the constraints of the mutated ones. The number of the mutated models having constraints violated by the test cases to the total number of the mutated models represents the effectiveness ability of these tests.

The question that it is investigated here is whether mutation analysis can provide a good indication about the quality of the test suites. A positive answer to this question will indicate that the proposed approach is valid and will motivate practitioners to use it. However, as already mentioned, CIT forms the mainstream approach to select and evaluate such test cases. Thus, another question that need to be answered is whether mutation analysis provides better estimations than the CIT about the fault detection ability of the test suites. Therefore, the main contribution of the present paper is the comparison of the proposed mutation approach with the CIT one according to their ability to expose faults. Currently, only a few works investigate the fault detection ability of the interaction testing in the presence of input constraints and none focusing on mutating the input models.

We present results of a controlled experiment that involves four real world programs with input constraints. The utilized programs are widely used in experimental studies and are accompanied by a set of faulty versions. Therefore, we can evaluate the ability of the examined approaches to predict the actual fault detection of the selected test suites. This is performed based on a rank correlation analysis. The findings of the study reveal that both the mutation and CIT approaches are good predictors of actual fault detection. This is in line with the previous research on CIT. However, it turns out the the mutation-based approach generally provides better estimations than CIT. This difference is significant in most of the cases, thus indicating a strong correlation between code-level faults and the proposed model-level defects.

In brief, the contributions of this paper are the following:

- We propose a mutation analysis approach applied at the program input level to assess the quality of test suites,
- We evaluate the correlation between a) the number of interactions covered and b) the number of the introduced mutants distinguished by a test suite with its actual fault detection. We find out that the model-based defects have a stronger correlation with code-level faults than the input parameter interactions.

II. Example

This section introduces an illustrative example to explain how mutation analysis can be applied. The approach is based on a model of the program inputs, as those typically used by CIT approaches, e.g., [4]. Such a model encompasses the different parameters and the constraints between these parameters.

A. An Input Model

Consider the following model M involving three parameters p_1, p_2 and p_3. Each parameter is a variable of the model. The parameter p_1 can take the two values a and b, p_2 can take the three values c, d and e and the last parameter p_3 can only take the f value. Thus, the model M is defined as follows:

$$M := p_1 \in \{a, b\},\ p_2 \in \{c, d, e\} \text{ and } p_3 \in \{f\}.$$

Typically, input models involve input constraints between the parameters. For sake of simplicity, we will first present the approach in the case where there are no input constraints, case 1). Then, the general case that involves input constraints, case 2), will be demonstrated.

B. Case 1: Absence of Input Constraints

1) Flattening the Model: In order to apply mutation analysis, we flatten this model to a boolean one denoted as M_b. The flattened model involves 6 variables instead of three, which corresponds to the values of the parameters: a_{p_1}, b_{p_1}, c_{p_2}, d_{p_2}, e_{p_2} and f_{p_3}. Each of this variable is boolean, i.e., it can take only two values, $true$ or $false$. We then need the to add to M_b constraints which specify that only one value can be selected at a time for a given parameter. For instance, a_{p_1} and b_{p_1} cannot be both $true$ because it would mean that $p_1 = a$ and $p_1 = b$ at the same time. Following our example, we need to add to M_b 8 following constraints:

$$(a_{p_1} \Rightarrow \neg b_{p_1}),\ (b_{p_1} \Rightarrow \neg a_{p_1}),\ (c_{p_2} \Rightarrow \neg d_{p_2}),\ (c_{p_2} \Rightarrow \neg e_{p_2}),$$
$$(d_{p_2} \Rightarrow \neg c_{p_2}),\ (d_{p_2} \Rightarrow \neg e_{p_2}),\ (e_{p_2} \Rightarrow \neg c_{p_2}),\ (e_{p_2} \Rightarrow \neg d_{p_2}).$$

Thus, the boolean model M_b equivalent to M is defined as:

$M_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $(e_{p_2} \Rightarrow \neg c_{p_2})$, $(e_{p_2} \Rightarrow \neg d_{p_2})$.

2) Creating Mutants of the Flattened Model: The next step consist in creating defective (i.e., mutated) versions of the M_b model. To produce such a mutated model, we alter one of the constraint of M_b. It creates a different version of this model where the defect is the change operated on the constraint. This process is repeated several times to create various mutated versions of M_b. For instance, the constraint $C = (a_{p_1} \Rightarrow c_{p_2})$ of M_b can be altered to $C' = (a_{p_1} \Rightarrow \neg c_{p_2})$ while producing a new mutated model from M_b.

In the following, we consider the two following mutants, in which the altered constraint of M_b is underlined:

- $M'_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$,

$(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $\underline{(e_{p_2} \Rightarrow c_{p_2})}$, $(e_{p_2} \Rightarrow \neg d_{p_2})$.

- $M_b'' := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $(e_{p_2} \Rightarrow \neg c_{p_2})$, $\underline{(e_{p_2} \Rightarrow f_{p_3})}$.

3) Evaluating Program Inputs: The proposed approach selects program inputs which satisfy the constraints of M_b and at the same time do not satisfy the constraints of the mutated models. For instance, consider the two following program inputs:

- $I_1 = \{a_{p_1} = true, b_{p_1} = false, c_{p_2} = true, d_{p_2} = false, e_{p_2} = false, f_{p_3} = true\}$,
- $I_2 = \{a_{p_1} = true, b_{p_1} = false, c_{p_2} = false, d_{p_2} = false, e_{p_2} = true, f_{p_3} = true\}$.

We simplify the representation of these inputs to consider only the values selected, i.e., equals to $true$:

- $I_1 = \{a_{p_1}, c_{p_2}, f_{p_3}\}$,
- $I_2 = \{a_{p_1}, e_{p_2}, f_{p_3}\}$.

Both I_1 and I_2 satisfy M_b. We evaluate each input towards each mutant. I_1 satisfies the two mutants, I_2 satisfies the second mutant M_b'' but violates M_b'. Indeed, the underlined constraint of M_b' is violated: selecting $e_{p_2} = true$ implies selecting $c_{p_2} = true$, but I_2 has $e_{p_2} = true$ and $c_{p_2} = false$. We thus identify that the I_2 test case is effective in finding the introduced defect. Measuring the number of such defects found by a test suite serves as an effectiveness measure to our approach. We can say that I_1 did not violated any of the two mutants and that I_2 violated half of the mutants. Thus, with respect to our approach, I_2 is more effective.

With respect to CIT, for instance 2-wise interactions, I_1 covers 15 interactions. An example of such an interaction is $(a_{p_1} = true, b_{p_1} = false)$. The total number of 2-wise interactions of the model M_b is 66. Thus, the CIT measure for I_1 is $\frac{15}{66}$. Consider now the two following test suites:

- $T_1 = \{I_2\}$,
- $T_2 = \{I_1, I_2\}$.

With respect to our approach, both the test suites are similarly effective since they both violates one of the two mutants. Thus, we measure $\frac{1}{2}$ for both T_1 and T_2, which is the number of violated mutants to the total ones.

With respect to CIT, T_1 covers $\frac{15}{66}$ 2-wise interactions while T_2 covers $\frac{24}{66}$. As a result, for CIT, the second test suite is more effective as it covers more interactions than the first one.

C. Case 2: Presence of Input Constraints

When there are input constraints in the model M, we simply transform them to boolean ones and we add them to the constraints of M_b. For instance, suppose M contains the input constraint $((p_1 = a) \Rightarrow (p_2 = c))$. It is transformed into $(a_{p_1} \Rightarrow c_{p_2})$. Thus, in this case, the boolean model of M is:

$M_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $(e_{p_2} \Rightarrow \neg c_{p_2})$, $(e_{p_2} \Rightarrow \neg d_{p_2})$, $\underline{(a_{p_1} \Rightarrow c_{p_2})}$.

The added input constraint is underlined. The process then continues similarly as the case 1), by mutating M_b (see Section II-B2).

III. Test Suite Evaluation

The global process of the proposed mutation approach is depicted by Figure 1. The technique operates on a model of the program inputs, presented in Section III-A, by creating defective (i.e., mutated) model versions. The application of the approach, detailed in Section III-C,is equivalent to CIT since it uses the same input models. However, instead of measuring the interactions covered by the test cases, as done by CIT (presented in Section III-B), it measures the ability of the test cases to distinguish the defective versions.

A. The Program Input Model

Applying CIT or the proposed approach requires building a model of the program inputs. This model represents the different test cases that can be derived by combining the program inputs. Thus, the model encompasses the different parameters of the program, their values, and the constraints that link them. It can be seen as a set of constraint, where each constraints involves variables (the parameters), their possible values and how they can be combined.

A constraint regulates the use of the input parameters. For instance, a constraint may denote that setting a specific parameter p_i with a specific value v prevents another parameter p_j from taking the value w. We can formalize this constraint

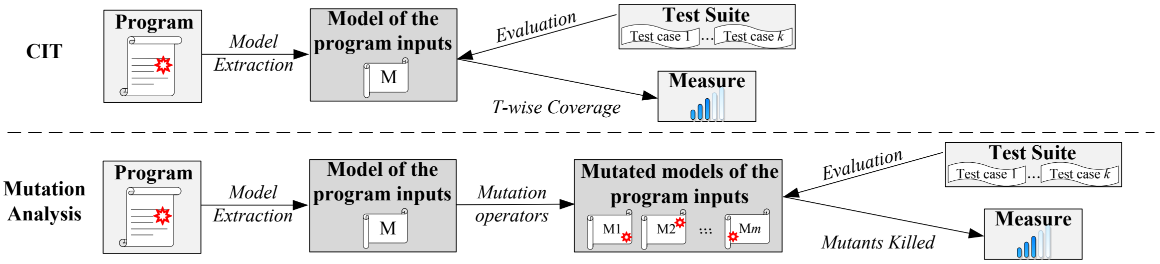


Fig. 1. The mutation analysis approach. The technique operates on a model of the program inputs by creating defective (i.e., mutated) model versions. The application of the approach is equivalent to CIT since it uses the same input model. However, instead of measuring the interactions covered by the test cases, as done by CIT, it measures the ability of the test cases to distinguish the defective versions.

TABLE I
The two mutation operators used to alter the constraints of the model of the program inputs. The first operator negates one of the variable of the constraint. The second operator splits the constraints into two by removing one of the disjunction operators.

Input	**Applies on**	**Result**
A constraint $C = v_1 \vee ... \vee \boldsymbol{v_i} \vee ... \vee v_k$	A variable v_i of the constraint	One constraint $C' = v_1 \vee ... \vee \neg\boldsymbol{v_i} \vee ... \vee v_k$
A constraint $C = v_1 \vee ... \vee v_i \vee ... \vee v_k$	A disjunction operator of the constraint	Two constraints: $C' = v_1 \vee ... \vee v_i$ and $C'' = v_{i+1} \vee ... \vee v_k$

as $(p_i = v) \Rightarrow (p_j \neq w)$. Thus, a program input composed of both $p_i = v$ and $p_j = w$ does not *satisfy the constraint* since p_j should not be set to w when p_i is set to v_i. We say that a test case *satisfies the model of the program inputs* if all the constraints of this model are satisfied at the same time.

Each constraint involves several variables that correspond to the parameters of the program. A variable has a domain which corresponds to the values that the parameter can take. For instance, the variable p_i may take two values v and w. In that case, p_i has a domain involving two values, v and w.

In practice, our approach requires a flattening of the model to make it boolean. This is a typical process undertaken by most of the CIT tools, e.g., [7]. It also gives the opportunity to the mutation approach to produce mutants in the case that no input constraints exists. Instead of having a model with variables corresponding to parameters, the flattened model contain variables that represent all the possible values for all the parameters. Each one of these variables can be *true* or *false*, depending on whether this variable is assigned an input value. For instance, if a parameter can take three different values, the flattening transforms the parameter variable, which has a domain of three values into three different variables. Doing so transforms the model to a boolean one as required by the proposed approach.

B. The CIT Approach

The CIT approach works by counting the unique number of interactions (or combinations) between any t parameters' values exercised by the test suite. Such interactions are called t-wise interactions, as they involve combinations between the t parameters. Thus, given the input model, we evaluate all the possible t combinations of the input parameters. Then, based on the input constraints, we eliminate the invalid ones, i.e., the combinations that are prohibited by the constraints. Finally, the effectiveness measure of the test suite is calculated based on the t-wise coverage which is the number of t combinations that are covered by the test cases.

C. The Mutation Approach

The mutation approach operates by altering the boolean model of the program inputs. It actually produces defective models by altering the model constraints. Thus, it creates several versions of the model, called *mutants*. Each mutant contains only one altered constraint. The constraints are altered based on a set of syntactic rules called *mutation operators*. Thus, by applying the mutation operators on all the model constraints, we end up with the sought set of mutants.

The constraints of the flattened model are boolean and they are represented as a disjunction between variables. Thus, each constraint C between k variables has the general form $C = \bigvee_{i=1}^{k} v_k$, where v_k is a variable (corresponding to one parameter's value) either set to *true* or *false*.

In this work, we employ two mutation operators adapted from [9]. These operators are presented in Table I. The first operator alters a constraint by taking one of its variables and negating it. In other words, if the selected variables is *true*, it becomes *false*, and conversely. The second operator aims at creating two constraints from the original one. To this end, one of the disjunction operator in the constraint is replaced by a conjunction operator. Thus, this second operator splits a constraint into two, increasing by one the total number of constraints of the model.

A test suite is evaluated based on its ability to distinguish the defective models from the original one. We refer to the mutants that are distinguished by the test cases as *killed* and to those that can not be distinguished as *live*. However, how can we check whether a mutant is killed or not? To answer this question, we need to consider that our models are composed of boolean constraints. Therefore, the evaluation is straightforward. It is examined whether a test case satisfies the constraints of the mutant model with a satisfiability (SAT) solver. The mutant is killed when its constraints are violated by the test case. In the opposite case it is live. It is noted that only the constraints of the mutant models can be violated. The constraints of the original model must always be satisfied in order to have valid program inputs. By determining the number of mutants that are killed by all the test cases, the overall effectiveness measure of the test suite is calculated.

IV. Experimental Methodology

The aims of the conducted experiment are summarized in the following research questions:

- **RQ1**: How well does the mutation-based approach evaluate the quality of the selected test suite?
- **RQ2**: How well does the CIT approach evaluate the quality of the selected test suite?
- **RQ3**: How does the mutation approach compares with CIT?

Answering the first question is important in order to substantiate the practical use of mutation. Answering the second question is important in reinforcing the empirical evidence in favor of CIT.

Now, suppose that mutation and CIT are capable of predicting accurately the fault detection ability of test suites. In this case, practitioners will be able to evaluate the quality of their test suites. Going a step further, they will be able to prioritize their tests (by pointing first the test cases that cover the majority of the interactions or kill most of the mutants), reduce the suites' size (by removing redundant tests) and guide the test generation process (by generating test cases that cover new interactions or kill additional mutants). However, in this case, which one should be used? This is our third research question which aims at identifying which of the two examined approaches should be used in practice.

A. Definition of the Experiment

To answer the stated research questions, we analyze the ability of test cases to cover t-wise interactions, to kill our mutants, and to expose code-based faults. We employ four subjects that are accompanied with test cases and faulty versions (Section IV-B). We then sample at random 30 test suites from the initial test suite of each program. Thus, we sample suites of random size from $4 \leq n \leq N$, where N is the size of the initial test suite. The minimum size of 4 test cases per suite has been chosen in order to have a sufficient sample for the correlation analysis. Then, we measure three metrics. a) the number of interactions covered, b) the number of mutants killed and c) the number of faults found (Section IV-C). These measures are recorded for every test case of the selected suites.

Then, these metrics are examined in order to identify possible correlations between them and to answer to RQs 1 and 2, Section IV-D. To this end, we perform a statistical analysis to quantify these correlations. In other words, we try to measure the extent to which the relationship of covering interactions and killing mutants relates with fault detection. Thus, for each subject program, 30 *Kendall rank coefficients* are obtained by evaluating the correlation between the killed mutants and the faults found by the test cases, case denoted as MF, and 30 coefficients are obtained from the correlation between the t-wise coverage and the faults found. This latter comparison is denoted as tWF. In this work, we consider t-wise coverage for $t = 2, ..., 4$.

Finally, we compare the two methods based on the level of correlations, thus, answering to RQ3. An overview of the followed process is depicted by Figure 2.

B. Subjects

We use the four following programs: `flex`, `gzip`, `make` and `sed`. These subjects are taken from the Software-artefact Infrastructure Repository (SIR) [10] and their details are presented in Table II. The examined versions of these program were randomly selected. Hence, for each subject, Table II records its size in uncommented lines of code [1], the number of faults per version taken from the faults matrix provided by the SIR, the number of variables and constraints of its respective model, the number of mutants and killable (i.e., there is at least one test configuration that cannot satisfy the faulty model) mutants obtained by applying the mutation operators, the number of the test cases contained in the initial test suite and the number of t-wise interactions for $t = 1, ..., 4$.

The input models are taken from the study of Petke et al. [4]. This study concerns the test case prioritization according to CIT and thus, their models are well suited for the present study. These models were built based on the descriptions of the Test Suite Specification Language (TSL) that are proposed with the utilized programs [4]. Thus, the parameters and values of these models represent the program input space. As described in Section III, we transform these models to boolean ones in order to evaluate the various test cases.

C. Evaluating Test Suites

1) T-wise: Given a test suite, we evaluate its t-wise coverage based on the following process. All the t-wise interactions between the parameters' values covered by the first test case of the suite are recorded. Then, we consider the second test case and we add all the interactions that are not exercised by the first one. This process is repeated for all the test cases of the suite and it gives the cumulative number of unique t-wise interactions covered by each one of suite' test cases. Thus, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the t-wise coverage represented by the tuples (tc_i, c_i).

2) Fault Detection: Given a test suite, we evaluate its fault detection ability based on the following process. We first take the faults found by the first test case of the suite by using the fault matrix provided by the SIR. This matrix contains the faults found by each test case. These faults are not considered while evaluating the next test cases. Then, the number of faults found by the second test case is recorded. This process is

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

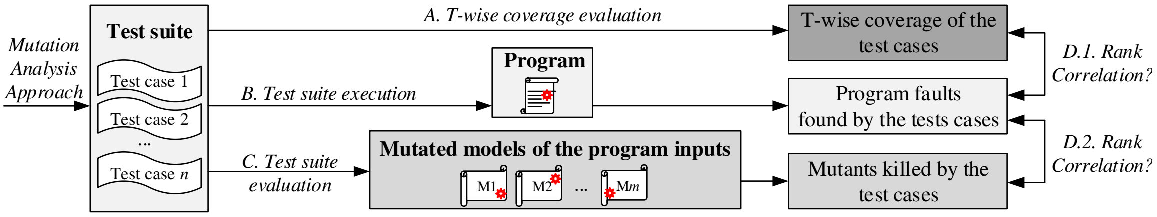

Mutation Analysis Approach
Test suite
Test case 1
Test case 2
...
Test case n
A. T-wise coverage evaluation
B. Test suite execution
C. Test suite evaluation
Program
Mutated models of the program inputs
M1
M2
...
Mm
T-wise coverage of the test cases
Program faults found by the tests cases
Mutants killed by the test cases
D.1. Rank Correlation?
D.2. Rank Correlation?

Fig. 2. The experimental methodology for comparing the mutation analysis approach with the CIT one. First, a test suite is generated by applying the mutation analysis approach. Then, three measures are evaluated: a) the t-wise coverage of the test suite, b) the number of faults found by the test suite by running the test cases on the program, and c) the number of mutants that are killed by the test suite. Finally, we evaluate 1) the correlation between the t-wise coverage and the faults found, and 2) the correlation between the mutants of the model killed ans the faults found.

TABLE II
THE FOUR SUBJECTS PROGRAMS USED IN THE EXPERIMENTS.

Subject		`flex`	`gzip`	`make`	`sed`
Uncommented lines of code	v1	9,581	4,604	14,459	-
	v2	-	5,092	-	-
	v3	-	-	-	7,161
	v4	11,470	-	-	-
	v7	-	-	-	14,177
Faults	v1	19	16	19	-
	v2	-	7	-	-
	v3	-	-	-	6
	v4	16	-	-	-
	v7	-	-	-	4
Model variables		23	29	20	34
Model constraints		43	91	21	143
Model mutants		139	295	63	527
Killable model mutants		139	292	63	373
Test cases		500	159	768	144
All 2-wise interactions		1,035	1,653	780	2,278
All 3-wise interactions		15,180	30,856	9,880	50,116
All 4-wise interactions		163,185	424,270	91,390	814,385
Valid 2-wise interactions		939	1,388	736	1,421
Valid 3-wise interactions		11,478	19,980	8,268	23,075
Valid 4-wise interactions		95,176	194,974	63,475	265,698

repeated for all the test cases of the test suite and provides the cumulative number of unique faults found by each test case. Thus, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the number of faults found after executing each test case. It is represented by the tuples (tc_i, f_i).

3) Mutation: We evaluate a test suite according to mutation based on the following process. Initially, the number of mutants killed by the first test case is determined. These mutants are removed and the second test case is evaluated according to all the remaining mutants. This process is repeated for the whole suite. Thus, the process gives the cumulative number of the unique killed mutants after executing each test case. Hence, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the number of mutants killed after executing each test case. It is represented by the tuples (tc_i, m_i).

D. Rank Correlation Analysis

Evaluating a test suite according to t-wise, fault detection and mutation, as described in the previous section gives the following information after executing $k \leq n$ tests of the test suite:

1) The current t-wise coverage achieved after considering the i^{th} test case of the test suite,
2) The current number of faults found after executing the i^{th} test case of the test suite,
3) The current number of mutated models that cannot be satisfied after considering the i^{th} test case of the test suite.

Given these three measures, we evaluate whether 1) correlates with 2), whether 3) correlates with 2), and which of these two correlations is better. In order to evaluate these correlations, we compute the *Kendall τ rank correlation coefficient*. This coefficient is considered as the most robust and usefully interpreted statistical measure for this question [11], [12], [13]. Thus, the coefficient is one one hand calculated given the correlation between the tuples (tc_i, c_i), (tc_i, f_i) and on the other hand given the tuples (tc_i, m_i) and (tc_i, f_i).

The Kendall τ is in the range $-1 \leq \tau \leq 1$. A coefficient of 1 indicates that the correlation between the two considered ranking is perfect. A coefficient around 0 denotes that the two observed sample are independent. Finally, a τ equals to -1 represents the complete absence of correlation between the two considered rankings.

V. EXPERIMENTAL RESULTS

This section reports results regarding the ability of the CIT and the mutation approaches to reveal actual faults. Then, it compares the two approaches.

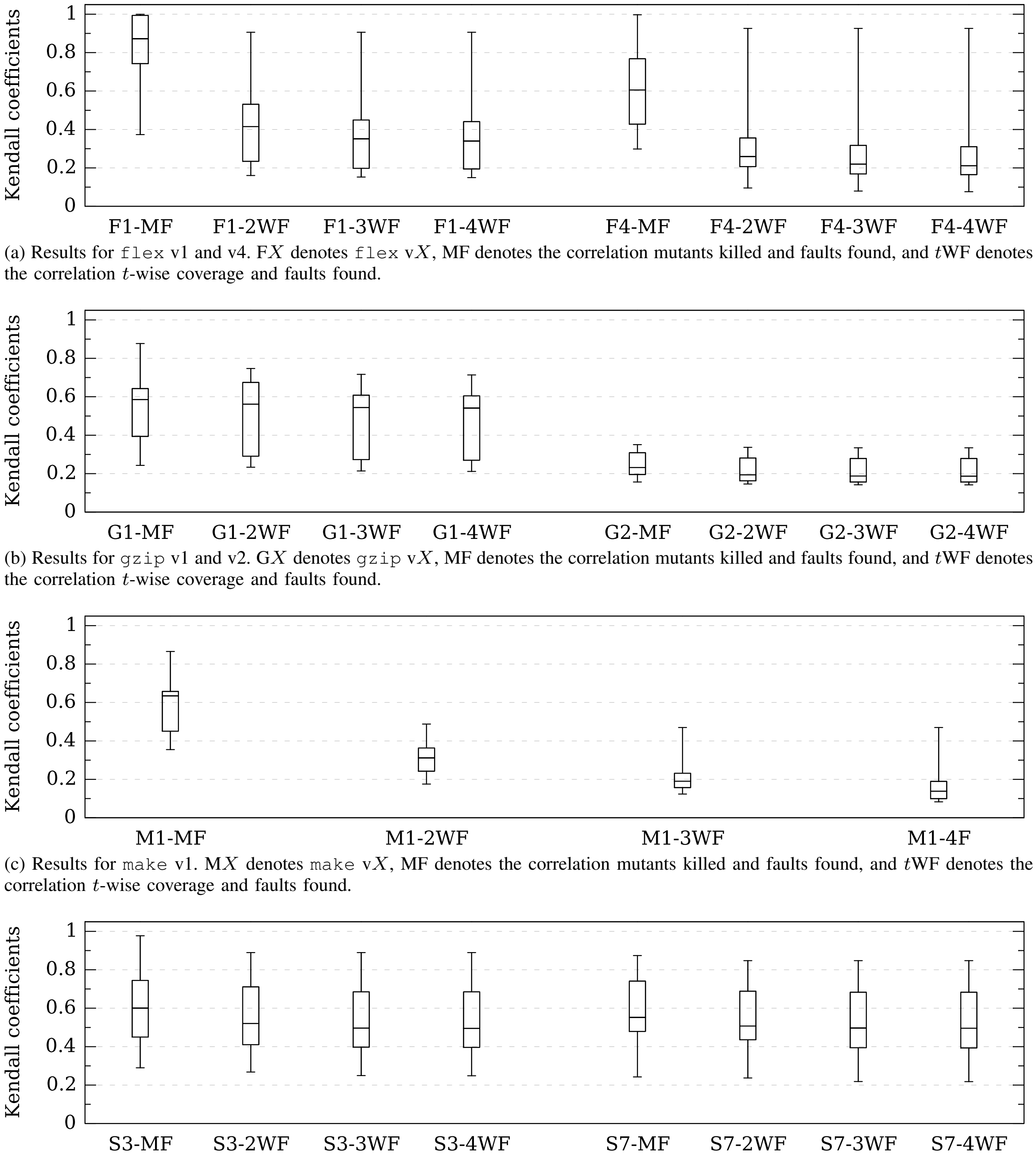
A. Correlation Analysis (RQ1 and RQ2)

Figure 3 presents the distribution the Kendall coefficients for all the subject programs. MF denotes the coefficients resulting from the correlation analysis between the mutants killed and the faults found by the test cases. The correlation coefficient between the t-wise coverage of the test cases and the faults found on the program are denoted as tWF, with $t = 2, ..., 4$.

From these results we can infer three interesting conclusions. First, we observe that both CIT and mutation strongly correlate with fault detection for all the employed subjects. This is due to the fact that most of the coefficients are greater than 0.5 (median values) and almost all are at least 0.3, thus indicating very good correlations. Second, we observe big differences on the correlations between the subject programs and between the different versions. For example, `sed` v3 has coefficients close to 0.5 while `gzip` v2 is close to 0.2. Similarly, `gzip` has differences between v1 and v2 where the coefficients are close to 0.5 and 0.2 respectively. Third, we observe that higher t-values results in weaker correlations than lower t values in all the examined cases. The next section compares t-wise and CIT based on these results.

B. Comparing CIT and Mutation (RQ3)

In order to compare the CIT approach with the mutation one, consider the MF and the tWF coefficients. From Figure 3, it is evident that MF coefficients tend to be closer to 1 than the tWF ones. Even when they are not very good, such as the case of `gzip` v2, they are greater than the tWF coefficients. The difference is big for three out of the 7 cases and always greater to all the tWF coefficients. This denotes a higher correlation



(a) Results for `flex` v1 and v4. FX denotes `flex` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.

(b) Results for `gzip` v1 and v2. GX denotes `gzip` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.

(c) Results for `make` v1. MX denotes `make` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.

(d) Results for `sed` v1 and v7. FX denotes `sed` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.

Fig. 3. Distribution of the Kendall τ rank coefficients on different versions of the subject programs. Each boxplot represents the distribution of the 30 τ coefficients of correlation between either the mutants killed and the faults found or between the t-wise ($t = 2, ..., 4$) coverage and the faults found.

between the mutants and the faults than between the t-wise coverage and the faults.

For instance, consider the results for `flex` v1, depicted by Figure 3a. The maximum coefficient τ is very close to 1. The median τ among the 30 coefficients is above 0.87. It Thus denotes a strong correlation for the MF case. Regarding the t-WF results, median coefficients are below 0.5, which denote moderate correlations between t-wise and the faults found by the test cases. Conclusively, it can be argued that the comparison is in favor of the MF since it gives correlations greater than the t-WF.

The results presented so far consider the correlation of CIT and mutation with the code-based faults. However, this correlation may be missleaded by the difficulty of finding the faults. In other words, if the faults are very easy to find or very difficult to find, this will have a direct effect on the measured correlations. In the same lines, we can compare the CIT and the mutation approaches.

Figure 4 presents the easiness of finding faults, killing mutants, covering 2-wise and covering 3-wise interactions

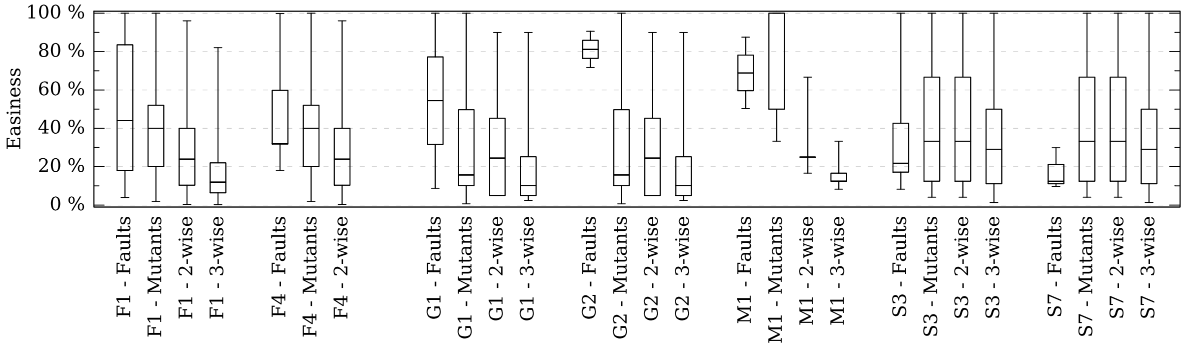


Fig. 4. The easiness of finding faults, killing mutants and covering t-wise interactions per subject. The easiness of finding a fault represents the percentage of test cases that find this fault. Similarly, the easiness of killing a mutant represents the percentage of test cases that kill this mutant, and the easiness of covering a t-wise interaction is the percentage of test cases that cover this interaction. FX, GX, MX, and SX respectively represents the `flex`, `gzip`, `make` and `sed` subjects, with the corresponding X version. For each subject, the fault box represents the easiness of the faults, the mutants box represent the easiness of the mutants and the t-wise box the easiness of the t-wise interactions.

per subject. The easiness of finding a fault represents the percentage of test cases that find this fault. Similarly, the easiness of killing a mutant represents the percentage of test cases that kill this mutant, and the easiness of covering a t-wise interaction is the percentage of test cases that cover this interaction. FX, GX, MX, and SX respectively represents the `flex`, `gzip`, `make` and `sed` subjects, with the corresponding X version. For each subject, the fault box represents the easiness for faults, the mutants box represent the easiness of the mutants and the t-wise box the easiness of the t-wise interactions.

From these results, we can explain why both CIT and mutation have low correlation values for `gzip` v2. This is due to the fact that the faults of this version are very easy to detect, contrary to mutants and interactions. It is the same for `make`. However, in this case, both mutants and interactions are easy to detect, resulting in satisfactory correlations.

Conclusively, from the presented results, it becomes evident that killing mutants is more or less as difficult as covering 2-wise and 3-wise interactions. Indeed, the easiness median values for mutants are comparable with the easiness medians of 2-wise and 3-wise interactions for `sed`, greater for `gzip` and lower for `flex` and `make`. Considering the easiness of mutants and faults, we observe that mutants tend to behave similarly as faults.

Finally, covering 3-wise interactions is harder than covering the 2-wise ones. This is expected since covering all 3-wise interactions results in covering all the 2-wise ones. As a result, higher interactions strengths ($t \geq 4$) are more difficult to cover.

VI. Discussion

The findings of the conducted experiment suggest that the mutation approach can form an alternative method to CIT. Based on the results, we can infer that mutation is probably more effective in predicting the actual fault detection of a test suite. In view of this, some additional considerations regarding the comparison with CIT and the application cost of the approach are needed. This section discusses these considerations as long as threats to the validity of the conducted experiment.

A. Additional Consideration about Mutation and CIT

By considering the CIT and the mutation approaches, we can observe that they more or less both use the same information, which is the input model. However, they provide much different results. Why this difference? In other words, why mutants can be more powerful than the input combinations?

Generally, it is very hard to fully answer this question. A full answer requires extensive and independent studies. However, we believe that the power of mutation lies on the fact that it considers the input constraints of the tested systems. Recall that our benchmark programs are real word programs. Thus, they have input constraints. These constraints play an important role in the testing process of the system. Not only they define the valid program inputs but they also reflect a logic of the underlying system. Therefore, they provide some useful information to the testers which is actually missed by CIT. Mutation takes advantage of this information by mutating these constraints. Mutating the input constraints forces tests to exercise limit cases that trigger faults. From the testing perspective, input constraints provide information similar to the one provided by boundary conditions of a system. They also have a role which is similar to the role of the preconditions of a system. Testing preconditions and boundary conditions of a system has been identified as an important step of the testing process [14]. Hence, testing them is necessary for establishing a rigorous testing approach. Here, it should be noted that CIT uses the input constraints only for computing the valid combinations of program inputs. Thus, it completely ignores both their importance and the information they provide.

Generally, mutation analysis relies on the power of the utilized mutants. The present studies uses two boolean operators mainly chosen based on the authors' experience from the feature modeling of software product lines [9], [15], [6]. Feature models are boolean models like the flattened ones used by the present study and hence, the employed operators form a good choice. In future, we plan to investigate the use of other operators, e.g., [16] and higher order ones [17], [16]. Nevertheless, the performed analysis shows that the utilized mutants simulate very well the behavior of the actual faults.

Here, it should be mentioned that the use of mutant selection strategies can increase the difficulty of finding them. However, the main question is whether doing so results in accurate estimations of the actual fault detection. Particularly, we do not want to underestimate or overestimate our measures [18]. This matter falls outside the scope of this paper since it is a general research challenge of the whole mutation testing area.

B. Cost of the Approach

One of the main issues of mutation analysis is its computational cost [19]. This is due to the need to introduce and execute a vast number of mutants. However, in our context the computational cost of mutation analysis is not very important due to the following three reasons. *First*, we only check whether a mutant violates the boolean constraints of the mutated models. This is a simple SAT verification process which is actually quite fast. *Second*, the number of mutants is small. Actually much smaller than the number of interactions, see Table II. For 2-wise, the number of mutants is 5 times lower than the number of interactions. As a result, mutation requires less operations than CIT. *Third*, we do not execute the system. Test selection and evaluation based on the input model is much faster than the actual program execution. Furthermore, in practice, testers will have to verify the program behavior, i.e., resolving the oracle problem, which is a typical manual activity. Hence, human time dominates the computational expenses of the approach.

Finally, it should be mentioned that equivalent mutants, i.e., mutants that can not be killed by any test case do not introduce a big overhead. Actually, the number of such mutants is much less than the number of invalid pairs (Table II). Therefore, the computational cost of mutation is lower than the cost of CIT.

C. Threats to Validity

There are several influencing factors that can threaten the validity of the conducted experiment. Regarding the generalization of the findings, i.e., *external validity*, it is possible that the selected programs are not representative. This may also be the case for the utilized test suites and the faulty versions. Thus, on larger or other types of programs, mutants and input interactions might not be the most appropriate choice. Similarly, the examined approaches might not being effective in revealing other faults. However, the chosen subjects are real world programs widely used in the literature e.g. [4], [20], [21]. Additionally, both the test suites and faults were developed by researchers independently of the present study. The problem we are facing is the absence of programs with high quality test cases, well defined input models or specifications, and faulty versions. Clearly, more studies are in need to answer this concern with confidence.

With respect to the confidence on the reported results, i.e., *internal validity*, issues on the utilized input models, the employed test sets and the correctness of the used tools can be identified. It is possible that errors on the input models and the used tools may have influenced the reported results. To reduce this threat we performed several manual checks on both the implementation and the employed input models. Here, it must be noted that the input models were independently developed by other researchers [4]. They were also checked by us in order to give confidence about their correctness. Additionally, as already mentioned, the employed suites are widely used in software engineering experiments e.g. [4], [20], [21].

Finally, some threats regarding the evaluation metrics used, i.e., *construct validity*, can be identified. It is likely that the number of faults found by the approaches do not express the real fault detection ability of the test suites. Additionally, it is possible that the faults number and difficulty can influence the significance of the performed statistical analysis. To reduce this threat, we employed the Kendall τ coefficient which is a non-parametric hypothesis test, i.e,. it does not require a very big sample size, and it measures the similarity of the data order when ranked by the studied effectiveness measures, i.e., the mutants found or the interactions covered. We also measured the correlations after executing every test case of various test suite sizes to eliminate the effects of the test suite size.

VII. Related Work

Mutation analysis is a powerful technique with multiple applications [8], [19]. Generally, code-based mutants have been used to guide the the test generation process [22], [23], [24], to assist the debugging activities [20], [25] and to evaluate the fault detection ability of a test suite [11], [18]. The technique has also been applied to test specification models [19] and to capture semantic errors of the programs [26]. For instance, Mottu *et al.* used mutation to test model transformations [27]. Other applications of this technique include Petri nets [28] and Feature Models [9], [15]. Contrary to these work, the present paper applies mutation analysis to the model of the program inputs and measures the correlation between model-based mutants and code-based faults.

CIT is a well researched technique with multiple criteria and combination strategies [1]. However, very few work consider the fault detection ability of CIT e.g. [4], [3], [5], [29], [30], [31]. In the most recent one, Petke *et al.* [4] show that higher t strengths result in finding more faults than lower strengths. Our work differs from this one in three ways. First, we use mutation while they only consider t-wise. Second, we consider the correlation between faults and t-wise interactions. They only investigate whether covering higher interaction strengths results in higher fault detection. Third, we use randomly selected test suites while they use test suites selected with a covering array tool [7]. Similarly, Arcuri and Briand [2] showed that random testing can perform similarly to CIT on large-scale models. However, their results hold only in the case where there are no input constraints.

Considering boolean specifications, mutation faults have been used to select minimum test suites [32]. Similarly, Kaminski *et al.* [16] use a logic mutation approach to measure test data quality. Their approach relies on the notion of higher order mutants [17] and aim at improving logic-based testing. In another work, Kaminski *et al.* [33] target at augmenting logic-based criteria inspired by the mutation approach. Contrary to these approaches, the present paper applies mutation on the program input model. Additionally, we measure the fault detection ability of this approach. Andrews *et al.* showed that generated mutants can be used to predict the detection effectiveness of real faults [18]. They investigate the relative cost and effectiveness of different testing coverage criteria. Here, we do not focus on whether or not the generated mutants of the model are representative of real defects.

Finally, the Kendall coefficient has been used in several work to measure the the correlation between two measured quantities. For instance, Gligoric *et al.* [11] performed a correlation analysis using this coefficient in order to evaluate the relationship between coverages and mutation score. In this work, we perform a correlation analysis by measuring the Kendall τ between a) the number of input parameter interactions covered by a given test suite and b) the number of the introduced mutants distinguished by the test suite with its actual fault detection.

VIII. Conclusion

In this paper, we proposed a mutation analysis approach as an alternative technique to CIT. We conducted a correlation analysis between a) the CIT and b) the mutation approach with their actual fault detection. Our results suggest that our mutants have a stronger correlation with code-level faults than the input interactions of the CIT approach. Therefore, mutation forms a valid measure of the test suites' quality.

Our future work includes experimentations with additional mutant operators, like replacing one parameter with another one. Additionally, possible ways of combining the examined techniques are also under investigation. Finally, we intend to broaden our study with additional programs and faults.

IX. Acknowledgment

The authors would like to thank Justyna Petke and Shin Yoo for for their help in understanding and setting the input models used by the present study.

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Softw. Test., Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[2] A. Arcuri and L. C. Briand, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Trans. Software Eng.*, vol. 38, no. 5, pp. 1088–1099, 2012.

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

[4] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *ESEC/SIGSOFT FSE*, 2013, pp. 26–36.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 633–650, Sep. 2008.

[6] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines," *CoRR*, vol. abs/1211.5451, 2012.

[7] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *ISSRE*, 2003, pp. 394–405.

[8] J. Offutt, "A mutation carol: Past, present and future," *Information & Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[9] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Assessing software product line testing via model-based mutation: An application to similarity testing," in *ICST Workshops*, 2013, pp. 188–197.

[10] 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.

[11] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *ISSTA*, 2013, pp. 302–313.

[12] M. G. Kendall, "A New Measure of Rank Correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, Jun. 1938.

[13] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. New-York, USA: Psychology Press, Sep. 1996.

[14] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *ISSTA*, 2011, pp. 364–374.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Towards automated testing and fixing of re-engineered feature models," in *ICSE*, 2013, pp. 1245–1248.

[16] G. K. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," *Information & Software Technology*, vol. 53, no. 10, pp. 1137–1152, 2011.

[17] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[18] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649 –678, sept.-oct. 2011.

[20] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, pp. n/a–n/a, 2013.

[21] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 3, pp. 19:1–19:29, July 2013.

[22] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, pp. 691–723, 2011.

[23] ——, "Mutation based test case generation via a path selection strategy," *Information & Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[24] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[25] M. Papadakis and Y. L. Traon, "Using mutants to locate ”unknown” faults," in *ICST*, 2012, pp. 691–700.

[26] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," *Sci. Comput. Program.*, vol. 78, no. 4, pp. 345–363, 2013.

[27] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Mutation analysis testing for model transformations," in *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ser. ECMDA-FA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 376–390.

[28] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong, "Mutation testing applied to validate specifications based on petri nets," in *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*. London, UK, UK: Chapman & Hall, Ltd., 1996, pp. 329–337.

[29] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empirical Software Engineering*, vol. 11, no. 4, pp. 583–611, 2006.

[30] L. Shi, C. Nie, and B. Xu, "A software debugging method based on pairwise testing," in *Proceedings of the 5th international conference on Computational Science - Volume Part III*, ser. ICCS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 1088–1091.

[31] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *DOSTA*, 2007, pp. 1–7.

[32] A. Gargantini and G. Fraser, "Generating minimal fault detecting test suites for general boolean specifications," *Information & Software Technology*, vol. 53, no. 11, pp. 1263–1273, 2011.

[33] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *J. Syst. Software*, 2012.