Efficient and Guaranteed Detection of t-way Failure-inducing Combinations

Paolo Arcaini  
National Institute of Informatics  
Tokyo, Japan  
arcaini@nii.ac.jp  

Angelo Gargantini  
DIGIP  
University of Bergamo  
Dalmine, Italy  
angelo.gargantini@unibg.it  

Marco Radavelli  
DIGIP  
University of Bergamo  
Dalmine, Italy  
marco.radavelli@unibg.it

Abstract—Combinatorial testing is a widely applied black-box testing technique, which is used to detect failures caused by parameter interactions (we call them failure-inducing combinations). Traditional combinatorial testing techniques provide fault detection, but most of them have weak fault diagnosis. In this paper, we propose a new fault characterization method called MiXTGeTe to locate all the failure-inducing combinations in a system under test, up to an interaction size decided by the user. Our method is based on adaptive black-box testing, in which test cases are generated based on outcomes of previous tests. We show that our method performs better than existing strategies that explore all the faults first, and then obtain the failure-inducing combination(s) for each failure.

Index Terms—Combinatorial testing, failure-inducing combination, failure diagnosis, test generation

I. INTRODUCTION

Experiments and industrial evidence suggest that software failures are usually caused by interactions among inputs or parameters of the system [12]. For this reason, combinatorial interaction testing (CIT), which consists in testing all the interactions of a given strength, is widely used and efficient in detecting bugs. By testing all the interactions till a given strength $t$, we can validate the system or discover if it contains parameter combinations that cause failure. An interaction of size $t$ (or $t$-way interaction) is an assignment of a specific value to each of the selected $t$ parameters. Although the size of combinations causing faults is almost always unknown, experiments show that generally even a low degree of interaction is enough to discover faults [13]. One of the main goal of CIT research is to find techniques that are able to cover all the interactions of a given strength with as few tests as possible. In this way, the faults can be found by running only a possibly small number of tests.

While test generation for CIT is a well-studied topic, fault detection and localization is still an open problem, although there are now some works targeting diagnosis and bug characterization [10]. When a failing test has been found, it remains unclear which combination in the failing test is responsible for the failure, since a test contains many combinations of different sizes. Knowing the interaction (and, therefore, also all the input configurations) that trigger failures is of help not only in correcting bugs, but also in understanding the impact of them. The particular interacting configuration that induces a failure, often directly reflects a use case scenario, which can be traced back from the input configuration. Knowing only the test that causes a fault, instead, often makes it impossible to trace back to the general use-case scenario (maybe involving several other input configurations) that caused the fault.

The problem is how to locate the combination that causes a failure when a fault is discovered [16], [17]. Indeed, in case of failure, there is a masking effect among the interactions [16] that makes hard the precise localization of the right combination. In order to avoid this masking effect, new tests are needed besides those necessary to cover all the interactions as required by CIT. Moreover, the test suite size optimization can play against fault localization: having each test to cover as many interactions as possible reduces the size of the test suite, but it may make the fault localization harder.

Classically, test generation and fault localization are done sequentially. First, a combinatorial test suite is generated and then executed against the real system. Then, if a fault has been found, new tests are built to try to locate the faulty combinations. This process is not very efficient (no information about failures is used during generation) and it generally does not guarantee to discover all the faulty combinations. Lately, there have been some approaches that interleave test generation, test execution, and fault localization [17]. Our approach follows this new trend and tries to efficiently build test suites taking into account possible test failures during test execution.

Most works do not guarantee to detect the real failure inducing combinations. Most approaches show that they are able to identify very suspicious combinations that are likely to be failure inducing [7], but no guarantee is given. However, under some precise assumptions, also testing can locate bugs. For instance, if one knows the maximum number and maximum strength of failure inducing interactions in advance [2], also particular combinatorial test suites statically generated (called locating arrays) can be used to identify those interactions. Our approach follows this trend as well: under some rather general assumptions, we are able to (dynamically) generate test suites that guarantee the detection of failure-inducing combinations.

The contribution of this paper is therefore twofold:
In order to model and manipulate combinatorial models, we use the tool CTWedge [6]. Note that a combinatorial model may also contain constraints that, however, are not considered in this work.

**Example 1.** Let’s consider the combinatorial model (originally proposed by Ghandehari et al. [9]) of the `totinfo` program from the Siemens Suite in the Software Infrastructure Repository (SIR) [3]; the model has \( n = 6 \) parameters, namely \( P = \{\text{tables, row, column, table_attribute, input_attribute, maxline}\} \), having enumerative values. Code 1 shows the representation of such model in CTWedge. In the following, for presentation purposes, we consider as running example a simpler combinatorial model \( M \) having 3 boolean parameters \( P = \{A, B, C\} \).

**Definition 1 (Combinatorial Model).** Let \( P = \{p_1, \ldots, p_m\} \) be the set of parameters. Every parameter \( p_i \) assumes values in the domain \( D_i = \{v_1^i, \ldots, v_{\alpha_i}^i\} \).

In order to model and manipulate combinatorial models, we use the tool CTWedge [6]. Note that a combinatorial model may also contain constraints that, however, are not considered in this work.

**Example 1.** Let’s consider the combinatorial model (originally proposed by Ghandehari et al. [9]) of the `totinfo` program from the Siemens Suite in the Software Infrastructure Repository (SIR) [3]; the model has \( n = 6 \) parameters, namely \( P = \{\text{tables, row, column, table_attribute, input_attribute, maxline}\} \), having enumerative values. Code 1 shows the representation of such model in CTWedge. In the following, for presentation purposes, we consider as running example a simpler combinatorial model \( M \) having 3 boolean parameters \( P = \{A, B, C\} \).

**Definition 2 (Test case).** Given a combinatorial model \( M \), a test case is an assignment of values to every parameter \( \{p_1, \ldots, p_m\} \) of \( M \). Formally, a test \( f \) is a \( m \)-tuple \( f = (p_1 = v_1, p_2 = v_2, \ldots, p_m = v_m) \), where \( v_i \in D_i \) for \( i \in \{1, \ldots, m\} \). We identify with \( f(p_i) \) the value \( v_i \) of parameter \( p_i \) in test \( f \). We use the function \( \text{result}(f) \) to indicate whether a test in a test suite \( TS \) passes or fails on a system \( SUT \):

\[
\text{result} : TS \rightarrow \{\text{pass, fail}\}
\]

i.e., \( \text{result}(f) \) is fail if and only if the test \( f \) fails, for example, because the SUT executed with \( f \) produces a wrong value or because an error or an exception occurs; \( \text{result}(f) \) is pass otherwise.

Note that other approaches [16] assume that different tests can fail in a different way, i.e., they can be distinguished by exception traces, state conditions, etc. In our setting, all the failing tests fail in the same way.

**Example 2.** Given the model \( M \) in Ex. 1, a possible test case is \( f = (A = 0, B = 1, C = 0) \). When a parameter is boolean, it can be denoted just with its name if its value is true (1), and with a bar above its name if its value is false (0). The example then becomes \( f = \overline{ABC} \).

**Definition 3 (Combination).** A combination (or partial test, or tuple or schema) \( c \) is an assignment to a subset \( \text{Dom}(c) \) of all the possible parameters \( P \), formally \( \text{Dom}(c) \subseteq P \). A test is thus a particular combination in which \( \text{Dom}(c) = P \). We identify with \( C_t \) the set of all the combinations of size \( t \) for a given set of parameters \( P \).

**Example 3.** For the model \( M \) introduced in Ex. 1, a possible combination is \( c = AB \).

**Definition 4 (Combination Containment).** A combination \( c_1 \) contains a combination \( c_2 \) if all the parameters of \( c_2 \) are also parameters of \( c_1 \), and their values are the same. Formally: \( \text{Dom}(c_2) \subseteq \text{Dom}(c_1) \land (\forall p_i \in \text{Dom}(c_2) : c_1(p_i) = c_2(p_i)) \).

**Example 4.** For instance, for the running example \( M \), the test \( f = \overline{ABC} \) contains the combination \( c = AB \).

**Definition 5 (Test Suite).** A test suite \( TS \) is a set of test cases. We denote with \( ETS \) the Exhaustive Test Suite that contains all the possible tests that can be formed from the specified combinatorial model; with \( CTS_t \), instead, we identify a Combinatorial Test Suite of strength (at least) \( t \), i.e., a test suite in which all the possible \( t \)-way interactions are covered by at least one test.

**Definition 6 (Failure-inducing combination).** A combination \( c \) is a failure-inducing combination (fic) for a test suite \( TS \) if each test that contains \( c \) fails. Formally, \( \forall f \in TS : c \subseteq f \rightarrow \text{result}(f) = \text{fail} \). We identify with \( \text{isFic}(c, TS) \) the predicate that tells whether the combination \( c \) is a failure inducing combination for a certain test suite \( TS \).

We call \( c \) a true-failure-inducing combination if we consider all the tests in the exhaustive test suite \( ETS \), i.e., if
isFic(c, ETS) holds. We call c a t-failure-inducing combination (ficc), if we consider all the tests in a CTS, i.e., if isFic(c, CTS) holds.

Observation 1. From Def. 6, we observe that a combination c is guaranteed not to be failure-inducing if there exists a test that contains it and does not fail.

Example 5. Let’s consider the model M introduced in Ex. 1 and the test suite shown in Table Ia. By definition, all the failing tests (# 3, 5, 6, 7, and 8) are failure-inducing combinations. In addition, we can notice that also the test suite shown in Table Ia is a failure-inducing combination, since all the tests containing them fail. Moreover, also the 1-way combination A is failure-inducing. In this example, the test suite is an exhaustive test suite, therefore these combinations are also true-failure-inducing combinations.

Definition 7 (Minimal failure-inducing combination). A failure-inducing combination c is minimal (mfic) if and only if all the combinations in c (except c itself) are not failure inducing in the test suite TS. Formally, isMfic(c, TS) if and only if isFic(c, TS) ∧ (∀ic’ < c: ¬isFic(c’, TS)). If we consider a combinatorial test suite CTS, we call c a t-minimal-failure-inducing combination (mfic).

Example 6. In the test suite shown in Table Ia, the combinations c1=A and c2=BC are both minimal.

III. DEFINITIONS

First we want to define when a failure-inducing combination has been located and isolated by a suitable test suite TS.

Definition 8 (Isolated mfic). An mfic c is isolated by a test f of a test suite TS if and only if c is the only fic in f, i.e.,

\[ \text{isIsoMfic}(c, f, TS) \equiv \text{isMfic}(c, TS) \land c \subseteq f \land \forall(c' \neq c) \in f: \neg \text{isMfic}(c', TS) \]

We say that a test suite TS isolates an mfic c iff

\[ \text{isIsoMfic}(c, TS) \equiv (\exists f \in TS: \text{isIsoMfic}(c, f, TS)) \]

Note that being able to isolate an mfic is particularly important for fault localization (that should follow our process); indeed, if two or more mfics are present in each failing test, it is more difficult to localize the fault as the mfics mask each other [16]. However, it is not always possible to isolate

### TABLE I: Test suites for running example

<table>
<thead>
<tr>
<th>test</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>fail</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>fail</td>
</tr>
</tbody>
</table>

### TABLE II: Test suites for running example

<table>
<thead>
<tr>
<th>test</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>fail</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
</tbody>
</table>

Each other. However, it is not always possible to isolate

mfics. Consider, for example, a SUT with boolean parameters \{A, B, C\}, whose true-mfics are AB, AC, and BC: AB cannot be isolated in this case.

**Theorem 1.** If the SUT has a true-mfic of strength t, in any combinatorial test suite CTS, there exists a failing test case.

**Proof.** By definition of combinatorial test suite.

A consequence of the theorem is the next corollary.

**Corollary 1.** If there is no failing test in a combinatorial test suite CTS, then there is no true-mfic of size t.

However, this property is not sufficient to isolate mfics of size t, as stated by the following theorem.

**Theorem 2** (Insufficient accuracy of CTS). A CTS does not guarantee to isolate mfics of size t.

**Proof.** Consider the running example whose true-mfics are A and BC. The combinatorial test suite of strength t=2 shown in Table Ib only has ABC and ABC as failing tests. The detected mfics are A, BC, and BC. We would need at least one more passing test containing BC to correctly classify it (test #2 in Table Ia). Moreover, in order to isolate BC, we would need one more test in which it fails alone (tests #3 in Table Ia).

IV. THE MixTgTE METHOD

Finding true-mfics can only be obtained using the exhaustive test suite ETS. However, exhaustive testing is in general not possible; therefore, we propose the approach MixTgTE (Mix Test Generation and Test Execution) that tries to identify and isolate mfics up to a given strength t. In order to do this, it uses combinatorial test suites CTS.

MixTgTE is an iterative process, as shown in Fig. 1 and Alg. 1. It starts from identifying combinations of size t=1 using the procedure MixTgTE1, and progressively repeats the search algorithm to combinations with higher size, until the user (who, at every iteration, can inspect the set ISOMFICS of discovered isolated mfics of size less or equal to t) decides to stop the process, or t reaches the number of parameters |P| of the system under test. The latter condition, however, is equivalent to exercising the exhaustive test suite, and it is normally infeasible in practice, except for trivial systems.
Algorithm 1 MiXTgTE

1: \text{ISOMFICS} \leftarrow \emptyset
2: \text{FT} \leftarrow \emptyset
3: \text{TS} \leftarrow \emptyset
4: t \leftarrow 1
5: \textbf{while} \ t \leq |P| \\textbf{and} User decides to continue \textbf{do}
6: \quad \text{MiXTgTE}(t, \text{ISOMFICS}, \text{TS})
7: \quad t \leftarrow t + 1
8: \textbf{end while}

Fig. 2: MiXTgTE_{t} process to find and isolate mfics up to accuracy of strength \( t \)

Algorithm 2 MiXTgTE_{t}

\textbf{Require}: \( t \): strength
\textbf{Require}: FT: isolated mfics computed at step \( t-1 \) (empty if \( t=1 \))
\textbf{Require}: TS: test suite computed at step \( t-1 \) (empty if \( t=1 \))
1: \text{FT} \leftarrow \{ c \subseteq f | f \in TS \land \text{result}(f) = \text{pass} \land c \in C_{t} \}
2: \text{FT} \leftarrow \text{FT} \cup \{ c \subseteq f | f \in TS \land \text{result}(f) = \text{fail} \land c \in C_{t} \} \setminus \text{PT}
3: \text{UT} \leftarrow C_{t} \setminus (\text{PT} \cup \text{FT})
4: \textbf{while} (\text{UT} = \emptyset \land (\text{FT} = \emptyset \lor (\forall c \in \text{FT}: \text{isExplained}(c, \text{TS})))) \textbf{do}
5: \quad f \leftarrow \text{buildTest}(\text{UT}, \text{FT})
6: \quad \text{TS} \leftarrow \text{TS} \cup \{ f \}
7: \quad \text{UPDATESETS}(f, \text{FT}, \text{PT}, \text{UT}, \text{ISOMFICS})
8: \quad \text{UPDATEMFICS}(\text{TS}, \text{FT}, \text{ISOMFICS})
9: \textbf{end while}

At each step, to keep limited the number of tests to execute on the SUT, the \textit{minimal} strength of the test suite used is equal to the size \( t \) of the detected combinations. Indeed, by Thm. 1, we can observe that this guarantees to have in the test suite all the \textit{mfics} of size \( t \). However, it could be some \textit{mfics} are not isolated; therefore, at each iteration, we also generate additional tests to isolate all the discovered \textit{mfics}

At each step, the user checks the returned sets of ISOMFICS to determine if it is the case to continue to search for \textit{mfics} of higher strength. The choice to continue or not is based on the available budget, but may also depend on the returned \textit{mfics} in ISOMFICS and the test suite TS.

A. MiXTgTE_{t}

Fig. 2 depicts the procedure MiXTgTE_{t} that, given a certain combination size \( t \), and a set of previously executed tests, produces a combinatorial test suite of strength \( t \) able to detect and isolate \textit{mfics} of strength up to \( t \). The process is described in detail in Alg. 2 and in the rest of the section.

MiXTgTE_{t} works on the following sets of combinations:
- \( UT \) (Unknown Tuples): the combinations of size \( t \) not appeared yet in any test during the process;
- \( PT \) (Passing Tuples): the combinations of size \( \leq t \) that were contained in at least one passing test executed so far in the process;
- \( FT \) (Failing Tuples): the combinations of size \( \leq t \) that were contained only by failing tests, among all the tests executed so far in the process, excluding the isolated \textit{mfics}.
- \( ISOMFICS \): the set of isolated \textit{mfics} detected so far (of size \( \leq t \)).

In addition, the process keeps track of the set of tests already run in the test suite \( TS \), together with the value of their result (either pass or fail).

We give a further definition that is used in the process.

\textbf{Definition 9 (Explained \textit{fic})}. Given the set ISOMFICS and a \textit{fic} \( c \in FT \), \( c \) is said to be explained if it implies one or more isolated \textit{mfics}, i.e.,

\[
\text{isExplained}(c, \text{ISOMFICS}) \equiv \exists S \in \mathcal{P}(\text{ISOMFICS}): c \rightarrow \bigwedge_{m \in S} m
\]

The rationale is that if a \textit{fic} \( c \) contains\(^1\) one or more iso-\textit{mfics}, the failure of the tests \( T_{c} \) in which \( c \) fails can be explained. Of course, this is just a heuristic, and some other test could show that actually \( c \) is the true \textit{mfic}. The definition of explained \textit{fic} will be used in the process to balance between \textit{exploitation} at strength \( t \) and \textit{exploration} of higher strengths.

1) \textbf{Tuple sets initialization}: Initially, \( PT \) contains all the tuples of size \( t \) that are contained in a passing test of \( TS \) (line 1); \( FT \), instead, inherits the failing tuples from previous iteration, and is enriched with \( t \)-tuples that are contained in a failing test but not in a passing test (line 2). \( UT \) is initialized with the remaining tuples of size \( t \) (line 3). ISOMFICS is kept from the previous step.

2) \textbf{Exit condition}: The process exits as soon as no unknown tuples \( UT \) are present, and either there are no failing tuples or all the failing tuples are explained (see Def. 9), i.e.,

\[
UT = \emptyset \land (FT = \emptyset \lor (\forall c \in FT: \text{isExplained}(c, TS)))
\]

3) \textbf{Test case generation}: If the exit condition is not met (i.e., there is at least an unknown tuple (UT) or a failing tuple (FT) that is not explained), the function buildTest generates a test \( f \) that contains at least one tuple belonging to either \( UT \), or to \( FT \) without being explained by any subset of \textit{mfics} in ISOMFICS. The generation works as shown in Alg. 3 and described as follows:

- if \( UT \) is not empty, we merge together as many tuples \( c \in UT \) as possible (lines 3-12). Two tuples \( c \) and \( c' \) cannot be merged if, for a given parameter \( p_{i} \), \( p_{i} \) has different values in \( c \) and \( c' \) (this is captured by predicate compatible at line 4). If after this phase some parameters

\(^1\)Note that, for conciseness, in the definition we use the propositional representation of tuples.
Algorithm 3 BUILDTEST: Test case generation

Require: TS: the tests generated so far
Require: UT: unknown tuples
Require: PT: failing tuples
1: f ← ∅
2: if UT ≠ ∅ then
3: for c ∈ UT do
4: if compatible(c, f) then
5: f ← f ∪ c
6: end if
7: if isCompleteTest(f) then
8: return f
9: end if
10: end for
11: f ← completeRnd(f)
12: return f
13: else
14: c_ac ← pickRnd(\{c ∈ TS | ¬isExplained(c, TS)\})
15: φ ← c_ac ∩ ∩\{f ∈ TS | c_ac ⊆ f\}
16: return getModel(φ) // A test is a satisfying assignment
17: end if

Algorithm 4 UPDATETUPLESETS: Tuple sets update

Require: f: a test
1: if result(f) = fail then
2: Move(f, UT, FT)
3: else
4: Move(f, UT, PT)
5: Move(f, FT, PT)
6: Move(f, ISOMFICS, PT)
7: end if
8: procedure Move(f, sourceSet, destSet)
9: toMove ← \{c ∈ sourceSet | c ⊆ f\}
10: sourceSet ← sourceSet \ toMove
11: destSet ← destSet ∪ toMove
12: end procedure

have no associated value, we randomly generate values for them (line 11);
• if instead UT is empty, we randomly select a not-
explained failing tuple c (line 14); then, in lines 15-16 we ask the SMT solver to find a test that contains c, but it is different from previous tests in TS (this is guaranteed to exist, as shown in Thm. 3).

4) Test execution and tuple sets update: After each test f is generated, it is immediately executed, and, depending on the result (pass/fail), the tuple sets are updated as described in Alg. 4:

1) If the test f fails, all combinations that are contained both in f and in the set UT, are moved from UT to FT;
2) If the test passes, we can exploit Obs. 1 and modify the sets as follows:
   a) all combinations that are contained both in f and in the set UT, are moved from UT to PT (line 4);
   b) all combinations that are contained both in f and in the set FT, are moved from FT to PT (line 5);
   c) all combinations that are contained both in f and in the set ISOMFICS, are moved from ISOMFICS to PT (line 6).

At this point, we can evaluate whether there are new isolated mfics, with the procedure shown in Alg. 5. If a tuple c ∈ FT turns out to be the only one (amongst all the possible tuples) to explain the failure of a test f (i.e., it is isolated in f according to Def. 8), it is added to ISOMFICS.

In summary, the status evolution of a combination c is depicted in the state machine shown in Fig. 3.

Example 7. On model M presented in Ex. 1, if the true-mfics were A and BC, a possible trace table of the process, with tests separated by the incremental maximum strength t, would be the one presented in Table II (the scenario with test 8a). We observe that the true-mfics have been correctly identified with the first two executions of MixTgTe (till test 6), i.e., tests 7 and 8a (for strength t=3) are not necessary, since the maximum strength of the true-mfics is 2.

Instead, if the true mfics were AB, AC, and AB, the process should be run three times for correctly identifying them (using test 8b), since there is a true-mfic of size 3.

V. PROPERTIES OF THE MIXTGTE PROCESS

In this section, we introduce some theorems assessing the capabilities of the proposed process.

We first make an assumption that is needed for our process.

Assumption 1. All true-mfics can be isolated.

Theorem 3 (Test case generation). In the test case generation (see Sect. IV-A3), it is always possible to generate a test case.

Proof. When UT is not empty, the test f is generated by merging compatible tuples from UT and then randomly selecting values for other parameters; since tuples in UT are those that have never been observed in any test, the new test f is guaranteed to exist. When UT is empty, the generated test must be an assignment satisfying formula φ at line 15 of Alg. 3. Let’s assume that such test does not exist; it would mean that all the possible tests T_cm containing c_mc have already been generated; there would be two cases:
TABLE II: Example of MIxTGtE for detecting mfics of different sizes with a strength up to \( t = 3 \)

<table>
<thead>
<tr>
<th>( # )</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
<th>ISOMFICS</th>
<th>( FT )</th>
<th>( PT )</th>
<th>( UT )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>pass</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{A, B, C}</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>fail</td>
<td>{}</td>
<td>{A, B, C}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>pass</td>
<td>{}</td>
<td>{A, B, C}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>fail</td>
<td>{A}</td>
<td>{A, B, C}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>fail</td>
<td>{A}</td>
<td>{A, B, C}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pass</td>
<td>{A, BC}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
<td>{AB, AC, BC, AB, AC, BC}</td>
<td>{}</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>fail</td>
<td>{A, BC}</td>
<td>{AB, AC, AB, AC, ABC, ABC, ABC}</td>
<td>{AB, AC, ABC, ABC}</td>
<td>{AB, AC, ABC, ABC}</td>
</tr>
<tr>
<td>8a</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>fail</td>
<td>{A, BC}</td>
<td>{AB, AC, AB, AC, ABC, ABC, ABC}</td>
<td>{AB, AC, ABC, ABC}</td>
<td>{}</td>
</tr>
<tr>
<td>8b</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>pass</td>
<td>{}</td>
<td>{AB, AC, ABC}</td>
<td>{AB, AC, ABC}</td>
<td>{}</td>
</tr>
</tbody>
</table>

1) If \( c \) is a true-mfic then ISOMFICS will contain \( c \). Let’s assume that \( c \) is a true-mfic but ISOMFICS does not contain it at the end. By Thm. 1, \( c \) is contained in a failing test of \( TS \) and so it is in \( FT \) at a given point. When the process terminates, \( FT \) is either empty (and so \( c \) is in ISOMFICS), or all the tuples in \( FT \) are explained (by Def. 9), i.e., they contain one or more iso-mfic. However, the latter case is not possible, as \( c \) would not be minimal.

2) If \( c \) is in ISOMFICS, it is a true-mfic. Let’s assume that \( c \) is in ISOMFICS, but it’s not a true-mfic. If \( c \) is in ISOMFICS, all tests containing it fail, and there exists a test in which it is the only mfic; if it is not a true-mfic, it means that in each test containing \( c \) there must be a combination \( c' \) such that \( c \subset c' \) and \( c' \) is a true-mfic and, therefore, added to ISOMFICS by point 1: in this case, \( c' \) would violate the minimality requirement. Note that, if \( c \) has size \( t \), there cannot be a true-mfic \( c' \) of higher strength containing \( c \) by Assumption 2.

VI. Evaluation

In this section, we evaluate the process and we compare it with other techniques for fault interaction detection.

A. Benchmarks

For the experiments, we selected some benchmarks, each one constituted by a faulty version of the SUT \( S_f \) and an oracle \( O \). The assessment of the execution of a test \( f \) (i.e., result in Def. 2) is performed by comparing the evaluations of \( f \) over \( S_f \) and \( O \). For practicality, we build a combinatorial model \( M \) having the same parameters of \( S_f \) and \( O \) agree (the constraints are the negation of the true-mfics). Therefore, for each benchmark, we also know the true-mfics in \( S_f \).

\({}\footnote{Note that \( S_f \) and \( O \) have the same parameters and they only differ on the behaviour.}
\footnote{Note that these constraints are not related to the combinatorial problem that, as stated in Sect. II, is unconstrained in our setting.}

Theorem 4 (Termination). The process is guaranteed to terminate.

Proof. The outer process MIxTGtE terminates when the user (i.e., the test engineer) decides not to continue it, or when \( t = |P| \).

The inner process MIxTGtE terminates when the exit condition (see Eq. 1) is met. The test generation phase (see Sect. IV-A3) directly aims at emptying \( UT \) and explaining all the not explained tuples in \( FT \). Since, by Thm. 3, the generation is always possible, the exit condition will be eventually met.

We want to prove that, under the assumption that the SUT has only true mfics of limited strength, by running the process till that strength, we will find them.

Assumption 2. Each true-mfic has maximum strength \( t \).

Theorem 5 (True-mfics found). If TRUE-MFICS is the set of true mfics and each \( c \) in TRUE-MFICS has maximum strength \( t \), then by running the process with strength equal or greater than \( t \), ISOMFICS is equal to TRUE-MFICS.

Proof. Under the stated assumption, the property is twofold: if \( c \) is a true-mfic, the MIxTGtE will find it and if the MIxTGtE finds a \( c \) as mfic, then \( c \) is a true-mfic.

1) If \( c \) is a true-mfic then ISOMFICS will contain \( c \). Let’s assume that \( c \) is a true-mfic but ISOMFICS does not contain it at the end. By Thm. 1, \( c \) is contained in a failing
We used two sets of benchmarks described in Table III. The first benchmark set, \textsc{BenchArt}, is constituted by artificial models of systems; we generated some of these models with one true-mfic (art1, art5, and art6), and others with multiple true-mfics. \textsc{BenchArt} also contains the running example, in its two versions shown in Table II. The second benchmark set, \textsc{BenchReal}, represents real systems: \texttt{aircraft} is a Software Product Line model presented in [19] and taken from the SPLIT repository\(^4\), and the others four are benchmarks used in Niu et al. [17].

In Table III, column size reports the size of model \(M\), presented in the abbreviated form \(k^{|\text{params}_k|} \times \ldots\), where \(k \in \mathbb{N}^+\) and \(\text{params}_k\) are the parameters having \(k\) values; for example, \(2^3 \times 1^4\) indicates that the SUT has 8 parameters that can take 2 values, one parameter taking 3 values, and one parameter taking 4 values. Column \(\text{TRUE-MFICS}\) reports the number \(|P|\) and size \(\text{size}\) of true-mfics in \(\mathcal{M}_f\); we report each possible size \(\text{size}\) with \(\text{size}(\#)\), where \(\text{size}(\#)\) denotes the set of \(\text{size}\) in parentheses. We also mark in bold face the maximum strength of the true-mfic; in the experiments, we assume Assumption 2, and so we apply the approach only up to the known maximal strength (according to Thm. 5, this guarantees to find all the mfics).

\begin{table}[h!]
\centering
\caption{Benchmark properties}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{name} & \textbf{|P|} & \textbf{size} & \textbf{TRUE-MFICS} \\
\hline
runExA & 3 & \(2^3\) & \((1(1), 2(1))\) \\
runExB & 3 & \(2^3\) & \((2(2), 3(1))\) \\
ar1 & 3 & \(2^3\) & \((2(1))\) \\
art2 & 3 & \(2^3\) & \((2(2))\) \\
ar3 & 3 & \(2^3\) & \((1(1), 2(1))\) \\
ar4 & 7 & \(2^7\) & \((1(1), 3(1))\) \\
ar5 & 7 & \(2^7\) & \((2(1))\) \\
ar6 & 5 & \(2^5 \times 2^1\) & \((3(1))\) \\
\hline
\textbf{BenchReal} & 8 & \(2^7 \times 3^1\) & \((3(1), 4(1))\) \\
tomcat & 12 & \(2^9 \times 3^1\) & \((1(1), 2(2))\) \\
hsqldb & 10 & \(2^9 \times 3^1\) & \((1(1), 3(2))\) \\
gcc & 10 & \(2^8 \times 3^1\) & \((3(4))\) \\
jflex & 13 & \(2^{10} \times 2^4\) & \((2(1))\) \\
\hline
\end{tabular}
\end{table}

\subsection{B. Compared approaches}

We compare our approach with some existing methods from literature, namely:

\textbf{BEN}: a process based on the first phase of the BEN tool proposed by Ghandehari et al. [7]. The process consists in calling BEN for failure-inducing combination detection, by providing an initial combinatorial test suite of a certain strength \(f\), and iterating over the size of the failure-inducing combinations to try to detect them. This process has already been used for constraints validation and repair [4], [5]. The BEN tool is included in our experimental process as a jar file.

\textbf{SOFOT}: the \textit{Simplified One Factor One Time} method to infer the Minimal Failure-causing Schema (MFS) from a given failing test case, from Nie et al. [15]. This method takes as input a set of failing tests, and tries to reduce each test to an mfc. For each failing test \(f\), it generates new tests by changing the value of each parameter in \(f\) one by one. Note that the source code of this method is available in Python from a later work by Zhang and Zhang [23]. As our automated evaluation script is written in Java, we program it so that it calls Python via command line. This causes some overhead which affects the total execution time in the experiments.

\textbf{FIC}: the \textit{Faulty Interaction Characterization} method proposed by Zhang et al. [23]. It is similar to SOFOT, in the sense that it accepts in input a set of tests known to be failing, and it tries to isolate the minimal failure-inducing combination(s) from it. It proceeds by considering one failing test a time, and changing the value of a parameter at a time, but, unlike SOFOT, it keeps the value changed. Furthermore, it performs a few iterations until the original failing test, with the value of the detected minimal failure-inducing combinations changed, passes. If there are two different failure-inducing combinations in the same tests, it may find them, but without a guarantee to be correct. We made a Java implementation of the algorithm described in the paper [23].

\textbf{ICT}: the \textit{Interleaving CT} approach proposed by Niu et al. [17]. It is a significant improvement of SOFOT that alleviates its three main problems: redundant test cases, multiple mfics, and masking effects (where multiple mfics are present in the same test). Like SOFOT, it is composed of two phases, generation and identification. Test generation is here made adaptive, one test at a time, in a similar way as the one of our approach. This reduces the amount of tests needed, by forbidding the generation of new tests containing already discovered failure-inducing combinations. The identification phase has a novel feedback checking mechanism (based on information coming from the execution of a few new proposed test cases), which can check, up to a certain extent, whether the identified mfc is a true-mfic or not; and it significantly improves the accuracy of the results w.r.t. SOFOT. The method is very recent, and, although we could not manage to re-run the tool on new benchmarks, we compared the results of \textsc{MixTgTgE} with the results of ICT reported in that paper for a common set of benchmarks.

Since FIC and SOFOT require failing tests as input, but do not say how to find such failing tests, we need to build a test suite to find such failing tests. In order to try to make the comparison fair, we use, for all the methods, an initial combinatorial test suite \(\text{CTS}_i\) of strength \(t=\max_{c \in \text{TRUE-MFICS}} \text{size}(c)\), being \(\text{TRUE-MFICS}\) the set of true-mfics (as shown in Table III). \(\text{CTS}_i\) is generated using \texttt{ACTS}\(^6\) for FIC, BEN, and SOFOT. \textsc{MixTgTgE}, instead, generates tests in an adaptive way, as described in Sect. IV-A3. ICT, instead, uses \texttt{AETG} [1].

In the following, \textit{DET-MFICS} denotes the set of mfics returned by a method; in our case, it corresponds to \textit{ISO-MFICS}.

\footnote{http://www.splot-research.org/}

\footnote{https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software}
C. Results

We run our method and the compared 4 methods 10 times for each benchmark; results are the average across the runs. Experiments have been executed on a Mac OS X 10.14, Intel Core i3, with 4GB of RAM. Code was written in Java, using CTWedge libraries for combinatorial modeling, test generation, and test execution [6]. The code and all the benchmarks are available online at [GitHub link].

Table IV shows the results of the experiments. For each method, it reports the total number of different tests required to complete the detection\(^7\), and the execution time in milliseconds. Moreover, in order to measure the quality of the returned mfics, we use classical measures as precision (P), recall (R), and F-score (F). Precision is defined as:

\[
\text{precision} = \frac{|\text{DET-MFICS} \cap \text{TRUE-MFICS}|}{|\text{DET-MFICS}|}
\]

Precision measures the percentage of found mfics that are true-mfics. If precision is not 1, the developer will spend some time in doing fault localization for a fic that is not a true-mfic (those in \(\text{DET-MFICS} \setminus \text{TRUE-MFICS}\)).

Recall is defined as:

\[
\text{recall} = \frac{|\text{DET-MFICS} \cap \text{TRUE-MFICS}|}{|\text{TRUE-MFICS}|}
\]

It measures how many true-mfics are actually identified. If the recall is not 1, the developer is not aware of a true-mfic that causes a fault (those in \(\text{TRUE-MFICS} \setminus \text{DET-MFICS}\)).

The F-measure is the combination of precision and recall, defined as follows:

\[
\text{F-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]

We now evaluate the approach answering the following three research questions.

RQ1: How is the effectiveness (in terms of precision and recall) of MixTGTec w.r.t. other techniques?

From the results presented in Table IV, we observe that MixTGTec always achieves maximum precision and recall; this is expected, as Thm. 5 guarantees that, under the assumption that we know the maximum strength \(t\), executing MixTGTec till strength \(t\) produces an ISOMFICS set (i.e., DET_MFICS) equal to TRUE_MFICS. All the other techniques do not provide this theoretical guarantee.

Among the other methods, ICT has the highest values for precision, recall, and F-score (92% on average) on the 4 available benchmarks [17]. For 3 benchmarks, ICT correctly identified all the TRUE_MFICS; only for gcc, some are wrongly identified (precision 77%) and some are not found (recall 65%).

Also FIC and SOFOT showed to be able to correctly identify the true-mfics in many occasions, although not with the same overall accuracy as ICT in terms of F-score (88% and 86%). We believe that this is due not only to the fixed amount of tests asked for the identification phase of those methods (they change always one parameter at a time, and only once), but also to the masking effect, i.e., when there are two mfics present in a same test. This effect may happen in general, as explained in Thm. 2. As an example of this fact, consider the running example described in Ex. 1 and the test suite generated by SOFOT shown in Table V. Let’s recall that the SUT is made of three binary parameters \(A, B, C\), with two true-mfics, \(A\) and \(BC\). By providing to SOFOT the faulty test cases observed with a combinatorial test suite of strength \(t = 2\) (that it is also the maximum strength of the true-mfics, so the correct settings for the experiments) generated with ACTS, the SOFOT method is able to correctly identify only the mfic \(A\). Table V reports, at the beginning, the \(CTS_2\) generated by ACTS; it contains two failing tests for which SOFOT tries to find the mfic. In test \(\dagger\), both true-mfics \(A\) and \(BC\) are contained; all the additional tests generated by SOFOT for this test (obtained by changing one parameter at a time) fail. Therefore, for test \(\dagger\), SOFOT does not find any mfic, i.e., it does not find \(A\) nor \(BC\). This is due to the masking effect in test \(\ddagger\) between \(A\) and \(BC\). The tests generated for the failing test \(\dagger\), instead, correctly identifies \(A\) as mfic.

\(\dagger\)If a test is generated twice by a method, we count it only once.
TABLE V: Execution trace of SOFOT on example SUT

<table>
<thead>
<tr>
<th>Generation of CTS with ACTS (to have some failing test)</th>
<th>Generation of CTS with ACTS (to have some failing test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>A</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Identification by SOFOT (tests added to find mftics) additional tests for failing test (1)

<table>
<thead>
<tr>
<th>#</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>fail</td>
</tr>
</tbody>
</table>

No mftic found additional tests for failing test (2)

<table>
<thead>
<tr>
<th>#</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>fail</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>fail</td>
</tr>
</tbody>
</table>

A identified as mftic

BEN is the method with the lowest F-score; this is because BEN is configured to produce fewer additional tests and uses heuristics to measure the suspiciousness of a failing tuple, and this may lead to wrong results.

RQ2: How does our approach compare with the others in terms of number of tests?

Overall, the number of tests required by MixTgTe is comparable to ICT. On the four real benchmarks in common, MixTgTe requires slightly fewer tests for gcc and jflex, but more tests for the other two benchmarks. For hsqldb, MixTgTe requires almost the double of the tests. This is due to the fact that ICT applies efficient heuristics to limit the number of tests that are asked in addition to the initial combinatorial test suite; MixTgTe, instead, does not have such strong optimizations, that we plan to investigate as future work. For this particular benchmark hsqldb, ICT is better (or equal) than our approach on any aspect (it also achieves 100% F-score); however, it does not provide any particular correctness guarantee.

SOFOT requires the highest amount of tests, and it obtains a lower recall, but a higher precision than FIC. The other two analyzed methods (FIC and BEN) require fewer tests (almost half of the test of MixTgTe on average), but, as described in RQ1, they also achieve less precision and recall than both MixTgTe and ICT.

RQ3: How does our approach compare with the others in terms of time?

All the reported times (for all the approaches) do not include the time for actually exercising the real system to determine the result (pass/fail) of the test. Indeed, the real system has been mocked by a model, since we know the true-mftics beforehand.

We cannot directly compare the execution time of ICT and SOFOT. Indeed, we were not able to rerun ICT on our machine (we report the results of the original paper [17]). For SOFOT, instead, we need to perform calls to an external Python program from Java, that introduce a big overhead.

The execution time for our process varies a lot depending on the number of generated tests, and the maximum strength achieved. It is less than 20ms for more than half of the benchmarks; however, it takes around 3.8 secs for hsqldb, which has two true-mftics of size 3, and one of size 1. The mftic of size 1 causes several tests to fail, masking the effect of the 3-way mftics. Note that, although gcc has four 3-way true-mftics, it takes less computation time because less tests are needed to isolate the mftics from the other failing tuples, as more tests are passing.

Generally, BEN is quite fast as it does not produce too many tests, and the time is not affected too much by the model size; in our case, instead, time is more dependent on the benchmark characteristics (model size, number of true-mftics, presence of masking effect, etc.).

FIC is the fastest method, as it only requires, on average, around 6ms per benchmark, with a maximum time of 36.6ms for hsqldb.

VII. RELATED WORK

Identifying the real failure inducing combinations is an area of active research in combinational testing [11], [15].

Previous works in detecting failure-inducing interactions are based on post-analysis of the test results of covering arrays (CAs), or on adaptive or non-adaptive test generation techniques. Yilmaz et al. [21] applied a post-analysis classification tree technique to analyze the result of CAs to find the differences between passing and failing tests. However, CA is not suitable to detect mftics precisely. Among non-adaptive methods, there is an approach based on pseudo-Boolean constraint solving and optimization, but its accuracy is highly affected by the chosen test suite [22]. Locating and detecting arrays (LDAs) [2], and error locating arrays (ELAs) [14] are other non-adaptive approaches: they require a given strength t and a maximum number of faulty interactions d, and they can detect and locate at most d faulty interactions of size up to t. However, the size of the test suite often becomes very large. That is why, recently, adaptive methods appear to be more studied in literature. They include Wang’s IterAIFL method [20], which is based on AIFL by Shi et al. [18], two adaptive algorithms proposed by Martinez et al. [14], and all the methods used to compare our process in the experiments: FIC (and also the variant FIC_BS) by Zhang et al. [23], BEN [8], SOFOT [15], and ICT [17].

While IterAIFL, FIC and SOFOT may not correctly identify multiple mftics in a system, since they may be overlapping or there is a masking effect, the two adaptive algorithms of Martinez work better but they can only locate mftics up to size 2. The ICT approach by Niu et al. [17], still derived from SOFOT, overcomes its limitations, making a significant improvement in the accuracy of the detected combinations. BEN [8] is tailored to locate faults in the code, but in the first
phase it provides an algorithm to detect suspicious combinations and, with some heuristics, failure-inducing combinations. However, as implemented so far, it is not very accurate with the initial test suites provided as input: an initial test suite of higher strength could improve accuracy of the detected mTICS. Unlike the other methods, MIXTgte does not distinguish between the two phases of the input test generation and additional adaptive test, but it merges those phases into one single process, that keeps track of the status of all the possible t-way tuples throughout the process. This way, MIXTgte has shown to correctly detect all the mTICS of a system, up to a certain strength \( t \) decided by the user, and it guarantees them to be correct under the assumption that there are no faults caused by an interaction of strength higher than \( t \).

VIII. CONCLUSIONS

The paper proposes an approach for finding minimal failure-inducing combinations (mTICS), that alternates test generation and test execution. Under the assumption that the maximum strength of true-mTICS is limited to \( t \), running the process till strength \( t \) guarantees to find all and only the true-mTICS; experimental comparison with state of the art approaches confirmed this fact. Achieving this total correctness does not affect too much the test suite size and the execution time: w.r.t. the second best approach (ICT) in terms of accuracy, MIXTgte produces slightly fewer tests in reasonable time.

The current work does not support constraints in the combinatorial model; their handling is planned as future work.

REFERENCES


