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 MIXTGTE 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:

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

```
Model totinfo

Parameters:
	tables: {t0, t1, t2}
	row: {r0, r1, r2}
	column: {c0, c1, c2}
	table_attribute: {ta0, ta1, ta2, ta3, ta4, ta5}
	input_attribute: {i0, i1, i2, i3, i4}
	maxline: {m1, m2, m3, m4, m5}
```

Code 1: A combinatorial model of the input of *totinfo* program, in CTWedge

1) detect and identify *all* the failure-inducing interactions in a system, up to a certain size t, if they exist;
2) doing it with a *smaller* test suite (compared to other techniques) by exploiting information coming during test execution about possible failure of tests.

During test generation, our approach collects tuples that seem to cause failures and tries to *isolate* them by finding those tuples that instead are *passing*. By using this information, tests are generated and immediately executed, and the knowledge base of the system updated accordingly.

The paper is structured as follows. Sect. II introduces the necessary background and Sect. III explains the definitions we need in our approach. Then, Sect. IV presents the iterative process we propose that combines test generation, test execution, and identification of failure inducing combinations. Sect. V discusses about the assumptions of the process and introduces some theorems about its capabilities. Finally, Sect. VI describes some experiments we performed to evaluate the process, Sect. VII reviews some related work, and Sect. VIII concludes the paper.

II. BACKGROUND

We assume that the software under test (SUT) has m input parameters that are described by a combinatorial model defined as follows.

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

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 m=6 parameters, namely P={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 $\{p_1, \dots, p_m\}$ of M. Formally, a test f is an m-tuple f = (p_1=v_1, p_2=v_2, …, p_m=v_m), where $v_i \in D_i$, for $i \in \{1, \dots, m\}$. We identify with $f(p_i)$ the value v_i of parameter p_i in test f. We use the function $result(f)$ to indicate whether a test in a test suite TS passes or fails on a system SUT:

$$result : TS \to \{pass, fail\}$$

i.e., $result(f)$ 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; $result(f)$ 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=$\bar{A}B\bar{C}$.

Definition 3 (**Combination**). A *combination* (or *partial test*, or *tuple* or *schema*) c is an assignment to a subset $Dom(c)$ of all the possible parameters P, formally $Dom(c) \subseteq P$. A test is thus a particular combination in which $Dom(c) = P$. 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=$\bar{A}B$.

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: $Dom(c_2) \subseteq Dom(c_1) \wedge (\forall p_i \in Dom(c_2)\colon c_1(p_i) = c_2(p_i))$.

Example 4. For instance, for the running example M, the test f=$\bar{A}B\bar{C}$ contains the combination c=$\bar{A}B$.

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, $\forall f \in TS\colon c \subseteq f \to result(f)$=*fail*. We identify with $isFic(c, TS)$ 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

TABLE I: Test suites for running example

(a) ETS

test	A	B	C	*result*
1	0	0	0	pass
2	0	0	1	pass
3	0	1	0	fail
4	0	1	1	pass
5	1	0	0	fail
6	1	0	1	fail
7	1	1	0	fail
8	1	1	1	fail

(b) CTS_2

test	A	B	C	*result*
7	1	1	0	fail
6	1	0	1	fail
4	0	1	1	pass
1	0	0	0	pass

$isFic(c, ETS)$ holds. We call c a t-failure-inducing combination (fic_t), if we consider all the tests in a CTS_t, i.e., if $isFic(c, CTS_t)$ 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 2-way combinations AB, $A\bar{B}$, AC, $A\bar{C}$, and $B\bar{C}$ are failure-inducing combinations, 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 $isFic(c, TS) \wedge (\forall c' \subset c \colon \neg isFic(c', TS))$. If we consider a combinatorial test suite CTS_t, we call c a t-minimal-failure-inducing combination (mfic_t).

Example 6. In the test suite shown in Table Ia, the combinations c_1=A and c_2=$B\bar{C}$ 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.,

$$isIsoMfic(c, f, TS) \equiv$$
$$isMfic(c, TS) \wedge c \subseteq f \wedge (\forall (c' \neq c) \subset f \colon \neg isMfic(c', TS))$$

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

$$isIsoMfic(c, TS) \equiv (\exists f \in TS \colon isIsoMfic(c, f, TS))$$

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

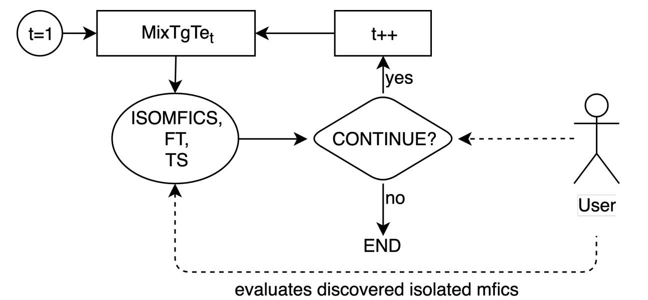

Fig. 1: Overview of the user-driven iterations of the process alternating test generation and detection of isolated mfics

mfics. Consider, for example, a SUT with boolean parameters $\{A, B, C\}$, whose true-mfics are AB, AC, and $B\bar{C}$: 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_t 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_t, 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_t)**.** *A CTS_t does not guarantee to isolate mfics of size t.*

Proof. Consider the running example whose true-mfics are A and $B\bar{C}$. The combinatorial test suite of strength t=2 shown in Table Ib only has $AB\bar{C}$ and $A\bar{B}C$ as failing tests. The detected mfics are A, $B\bar{C}$, and $\bar{B}C$. We would need at least one more passing test containing $\bar{B}C$ to correctly classify it (test #2 in Table Ia). Moreover, in order to isolate $B\bar{C}$, 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_t.

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 MixTgTe$_t$, 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

$ISOMFICS \leftarrow \emptyset$
$FT \leftarrow \emptyset$
$TS \leftarrow \emptyset$
$t \leftarrow 1$
while $t \leq |P| \wedge$ User decides to continue **do**
  MIXTGTE$_t$(t, $ISOMFICS$, TS)
  $t \leftarrow t + 1$
end while

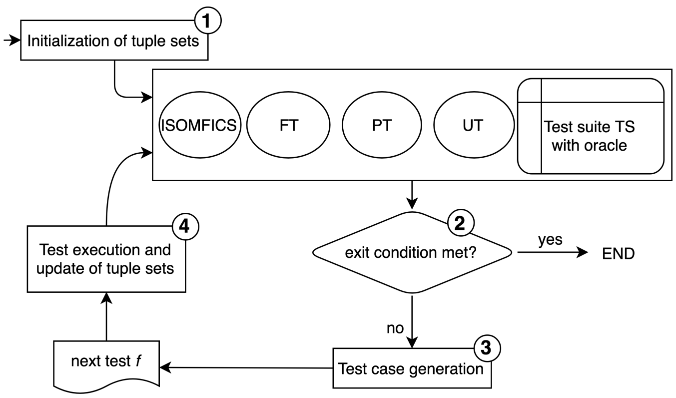

Fig. 2: MIXTGTE$_t$ process to find and isolate mfics up to accuracy of strength t

Algorithm 2 MIXTGTE$_t$

Require: t: strength
Require: $ISOMFICS$: isolated mfics computed at step t-1 (empty if t=1)
Require: FT: failing tuples found at step t-1 (empty if t=1)
Require: TS: test suite computed at step t-1 (empty if t=1)
1: $PT \leftarrow \{c \subseteq f | f \in TS \wedge result(f) = pass \wedge c \in C_t\}$
2: $FT \leftarrow FT \cup \{c \subseteq f | f \in TS \wedge result(f) = fail \wedge c \in C_t\} \setminus PT$
3: $UT \leftarrow C_t \setminus (PT \cup FT)$
4: **while** $\neg(UT = \emptyset \wedge (FT = \emptyset \vee (\forall c \in FT\colon isExplained(c, TS))))$ **do**
5:   $f \leftarrow buildTest(UT, FT)$
6:   $TS \leftarrow TS \cup \{f\}$
7:   UPDATETUPLESETS($f, PT, FT, UT, ISOMFICS$)
8:   UPDATEMFICS($TS, FT, ISOMFICS$)
9: **end while**

At each step, to keep limited the number of tests to execute on the SUT, the *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 mfics of size t. However, it could be some mfics are not isolated; therefore, at each iteration, we also generate additional tests to isolate all the discovered mfics.

At each step, the user checks the returned sets of $ISOMFICS$ to determine if it is the case to continue to search for mfics of higher strength. The choice to continue or not is based on the available budget, but may also depend on the returned 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 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 mfics.
- $ISOMFICS$: the set of isolated 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.

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

$$\begin{aligned} & isExplained(c, ISOMFICS) \equiv \\ & \exists S \in \mathcal{P}(ISOMFICS)\colon c \rightarrow \bigwedge_{m \in S} m \end{aligned}$$

The rationale is that if a fic c contains[1] one or more iso-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 mfic. The definition of explained fic will be used in the process to balance between *exploitation* at strength t and *exploration* of higher strengths.

1) 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) 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.,

$$UT = \emptyset \wedge (FT = \emptyset \vee (\forall c \in FT\colon isExplained(c, TS))) \quad (1)$$

3) 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 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 in 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: FT: failing tuples
1: $f \leftarrow \emptyset$
2: **if** $UT \neq \emptyset$ **then**
3:   **for** $c \in UT$ **do**
4:     **if** $\mathit{compatible}(c, f)$ **then**
5:       $f \leftarrow f \cup c$
6:     **end if**
7:     **if** $\mathit{isCompleteTest}(f)$ **then**
8:       **return** f
9:     **end if**
10:   **end for**
11:   $f \leftarrow \mathit{completeRnd}(f)$
12:   **return** f
13: **else**
14:   $c_{ne} \leftarrow \mathit{pickRnd}(\{c \in FT | \neg \mathit{isExplained}(c, TS)\})$
15:   $\phi \leftarrow c_{ne} \wedge \bigwedge_{\{f \in TS | c_{ne} \subseteq f\}} \neg f$
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** $\mathit{result}(f) = \mathit{fail}$ **then**
2:   MOVE(f, UT, FT)
3: **else**
4:   MOVE(f, UT, PT)
5:   MOVE(f, FT, PT)
6:   MOVE(f, $\mathit{ISOMFICS}$, PT)
7: **end if**

8: **procedure** MOVE(f, $\mathit{sourceSet}$, $\mathit{destSet}$)
9:   $\mathit{toMove} \leftarrow \{(c \in \mathit{sourceSet}) | c \subseteq f\}$
10:   $\mathit{sourceSet} \leftarrow \mathit{sourceSet} \setminus \mathit{toMove}$
11:   $\mathit{destSet} \leftarrow \mathit{destSet} \cup \mathit{toMove}$
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 $\mathit{ISOMFICS}$, are moved from $\mathit{ISOMFICS}$ 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 $c \in FT$ turns out to be the only one (amongst all the possible tuples) to

Algorithm 5 UPDATEMFICS: $\mathit{ISOMFICS}$ set update

Require: TS: the tests generated so far
Require: $\mathit{ISOMFICS}$: isolated mfics
Require: FT: failing tuples
1: **for** $c \in FT$ **do**
2:   **if** $\mathit{isIsoMfic}(c, TS)$ **then**
3:     $FT \leftarrow FT \setminus \{c\}$
4:     $\mathit{ISOMFICS} \leftarrow \mathit{ISOMFICS} \cup \{c\}$
5:   **end if**
6: **end for**

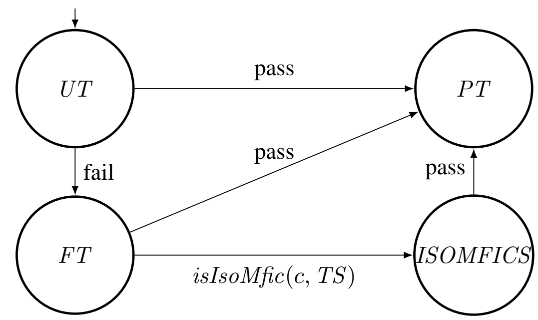

Fig. 3: Status evolution of a tuple c throughout the process

explain the failure of a test f (i.e., it is isolated in f according to Def. 8), it is added to $\mathit{ISOMFICS}$.

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 $B\bar{C}$, 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$_t$ (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 $A\bar{B}$, AC, and $\bar{A}B\bar{C}$, 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 $T_{c_{ne}}$ containing c_{ne} 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 $t = 3$

	#	A	B	C	*result*	*ISOMFICS*	*FT*	*PT*	*UT*
$t=1$	Fill FT-PT-UT					$\{\}$	$\{\}$	$\{\}$	$\{A,B,C,\bar{A},\bar{B},\bar{C}\}$
	1	0	0	0	pass	$\{\}$	$\{\}$	$\{\bar{A},\bar{B},\bar{C}\}$	$\{A,B,C\}$
	2	1	1	1	fail	$\{\}$	$\{A,B,C\}$	$\{\bar{A},\bar{B},\bar{C}\}$	$\{\}$
	3	0	1	1	pass	$\{\}$	$\{A\}$	$\{\bar{A},\bar{B},\bar{C},B,C\}$	$\{\}$
	updatedMfics					$\{A\}$	$\{\}$	$\{\bar{A},\bar{B},\bar{C},B,C\}$	$\{\}$
$t=2$	Fill FT-PT-UT					$\{A\}$	$\{AB,AC\}$	$\{\bar{A}\bar{B},\bar{A}\bar{C},\bar{B}\bar{C},\bar{A}B,\bar{A}C,BC\}$	$\{A\bar{B},A\bar{C},\bar{B}C,B\bar{C}\}$
	4	1	0	0	fail	$\{A\}$	$\{AB,AC,A\bar{B},A\bar{C}\}$	$\{\bar{A}\bar{B},\bar{A}\bar{C},\bar{B}\bar{C},\bar{A}B,\bar{A}C,BC\}$	$\{B\bar{C},\bar{B}C\}$
	5	0	1	0	fail	$\{A\}$	$\{AB,AC,A\bar{B},A\bar{C},B\bar{C}\}$	$\{\bar{A}\bar{B},\bar{A}\bar{C},\bar{B}\bar{C},\bar{A}B,\bar{A}C,BC\}$	$\{\bar{B}C\}$
	updatedMfics					$\{A,B\bar{C}\}$	$\{AB,AC,A\bar{B},A\bar{C}\}$	$\{\bar{A}\bar{B},\bar{A}\bar{C},\bar{B}\bar{C},\bar{A}B,\bar{A}C,BC\}$	$\{\bar{B}C\}$
	6	0	0	1	pass	$\{A,B\bar{C}\}$	$\{AB,AC,A\bar{B},A\bar{C}\}$	$\{\bar{A}\bar{B},\bar{A}\bar{C},\bar{B}\bar{C},\bar{A}B,\bar{A}C,BC,\bar{B}C\}$	$\{\}$
$t=3$	Fill FT-PT-UT					$\{A,B\bar{C}\}$	$\{AB,AC,A\bar{B},A\bar{C},ABC,\bar{A}B\bar{C},A\bar{B}\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C},\bar{A}BC,\bar{A}\bar{B}C\}$	$\{A\bar{B}C,AB\bar{C}\}$
	7	1	0	1	fail	$\{A,B\bar{C}\}$	$\{AB,AC,A\bar{B},A\bar{C},A\bar{B}C,A\bar{B}\bar{C},ABC,\bar{A}B\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C},\bar{A}BC,\bar{A}\bar{B}C\}$	$\{AB\bar{C}\}$
	a) Scenario in which test 8 fails								
	8a	1	1	0	fail	$\{A,B\bar{C}\}$	$\{AB,AC,A\bar{B},A\bar{C},A\bar{B}C,A\bar{B}\bar{C},ABC,\bar{A}B\bar{C},AB\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C},\bar{A}BC,\bar{A}\bar{B}C\}$	$\{\}$
	b) Scenario in which test 8 passes								
	8b	1	1	0	pass	$\{\}$	$\{A\bar{B},AC,A\bar{B}C,A\bar{B}\bar{C},ABC,\bar{A}B\bar{C}\}$	$\{A,B\bar{C},AB,A\bar{C},\bar{A}\bar{B}\bar{C},\bar{A}BC,\bar{A}\bar{B}C,AB\bar{C}\}$	$\{\}$
	updatedMfics					$\{A\bar{B},AC,\bar{A}B\bar{C}\}$	$\{A\bar{B}C,A\bar{B}\bar{C},ABC\}$	$\{A,B\bar{C},AB,A\bar{C},\bar{A}\bar{B}\bar{C},\bar{A}BC,\bar{A}\bar{B}C,AB\bar{C}\}$	$\{\}$

- at least one of the tests in $T_{c_{ne}}$ passes; this is not possible, as, in this case, c_{ne} would be in PT;
- all the tests in $T_{c_{ne}}$ fail; also this is not possible, as, in this case, c_{ne} would be either in *ISOMFICS* or explained by a subset of tuples in *ISOMFICS*.

□

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$_t$ 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 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[2] and constraints that accept only the tests for which S_f and O agree (the constraints are the negation of the true-mfics).[3] Therefore, for each benchmark, we also know the true-mfics in S_f.

[2]Note that S_f and O have the same parameters and they only differ on the behaviour.

[3]Note that these constraints are not related to the combinatorial problem that, as stated in Sect. II, is unconstrained in our setting.

TABLE III: Benchmark properties

	name	$\|P\|$	size	$TRUE\_MFICS$ (size (#))
$\mathtt{BENCH}_{\mathtt{ART}}$	`runExA`	3	2^3	1(1), **2**(1)
	`runExB`	3	2^3	2(2), **3**(1)
	`art1`	3	2^3	**2**(1)
	`art2`	3	2^3	**2**(2)
	`art3`	3	2^3	1(1), **2**(1)
	`art4`	7	2^7	2(1), **3**(1)
	`art5`	7	2^7	**2**(1)
	`art6`	5	$2^3 3^1 5^1$	**3**(1)
$\mathtt{BENCH}_{\mathtt{REAL}}$	`aircraft`	8	$2^7 3^1$	3(1), **4**(1)
	`tomcat`	12	$2^8 3^1 4^1$	1(1), **2**(2)
	`hsqldb`	10	$2^9 6^1$	1(1), **3**(2)
	`gcc`	10	$2^8 3^1 4^1$	**3**(4)
	`jflex`	13	$2^{10} 3^2 4^1$	**2**(1)

We used two sets of benchmarks described in Table III. The first benchmark set, $\mathtt{BENCH}_{\mathtt{ART}}$, 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. $\mathtt{BENCH}_{\mathtt{ART}}$ also contains the running example, in its two versions shown in Table II. The second benchmark set, $\mathtt{BENCH}_{\mathtt{REAL}}$, represents real systems: `aircraft` is a Software Product Line model presented in [19] and taken from the SPLOT 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 $k^{\#params_k} \times \ldots$, where $k \in N^+$ and $params_k$ are the parameters having k values; for example, $2^8 3^1 4^1$ indicates that the SUT has 8 parameters that can take 2 values, one parameter taking 3 values, and one parameter taking 4 values. Column $TRUE\_MFICS$ reports the number and size of true-mfics in M_f; we report each possible size with the number of mfics of that 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).

B. Compared approaches

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

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 t, 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.

SOFOT: the *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 mfic. 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.

FIC: the *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].

ICT: the *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 mfic 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 rerun the tool on new benchmarks, we compared the results of MIXTGTE 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[5], an initial combinatorial test suite CTS_t of strength $t = \max_{c \in TRUE\_MFICS} |size(c)|$, being $TRUE\_MFICS$ the set of true-mfics (as shown in Table III). CTS_t is generated using ACTS[6] for FIC, BEN, and SOFOT. MIXTGTE, instead, generates tests in an adaptive way, as described in Sect. IV-A3. ICT, instead, uses AETG [1].

In the following, DET_MFICS denotes the set of mfics returned by a method; in our case, it corresponds to $ISOMFICS$.

[4]http://www.splot-research.org/

[5]Note that MIXTGTE, BEN, and ICT already require a combinatorial test suite.

[6]https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software

TABLE IV: Experimental results (P: precision, R: recall, F: F-score, time is in ms)

	model	MIXTGTE					FIC					BEN					SOFOT					ICT				
		tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time
$\mathrm{BENCH}_{\mathrm{ART}}$	runExA	7.6	1	1	1	15.9	8	1.00	1.00	1.00	0.4	7	0.20	0.50	0.29	53.2	8	0.75	0.50	0.58	518	–	–	–	–	–
	runExB	8.0	1	1	1	4.9	8	0.75	1.00	0.86	0.6	8	0.25	0.33	0.29	9.4	8	0.75	1.00	0.86	735	–	–	–	–	–
	art1	6.6	1	1	1	3.2	5	1.00	1.00	1.00	0.5	7	1.00	1.00	1.00	13.0	7	1.00	1.00	1.00	269	–	–	–	–	–
	art2	8.0	1	1	1	5.4	7	1.00	1.00	1.00	0.6	7	0.50	0.50	0.50	13.8	8	0.50	0.50	0.50	396	–	–	–	–	–
	art3	7.6	1	1	1	2.7	8	1.00	1.00	1.00	0.7	7	0.00	0.00	–	15.1	8	1.00	0.50	0.67	403	–	–	–	–	–
	art4	36.9	1	1	1	79.7	29	0.67	1.00	0.80	1.8	26	0.00	0.00	–	64.8	61	1.00	1.00	1.00	2772	–	–	–	–	–
	art5	14.7	1	1	1	5.0	12	1.00	1.00	1.00	0.5	18	1.00	1.00	1.00	14.1	20	1.00	1.00	1.00	769	–	–	–	–	–
	art6	45.4	1	1	1	17.1	35	0.50	1.00	0.67	3.2	38	1.00	1.00	1.00	21.1	52	1.00	1.00	1.00	1830	–	–	–	–	–
$\mathrm{BENCH}_{\mathrm{REAL}}$	aircraft	89.8	1	1	1	156.5	71	1.00	1.00	1.00	18.4	62	0.00	0.00	–	56.4	115	1.00	1.00	1.00	4157	–	–	–	–	–
	gcc	88.5	1	1	1	134.5	64	0.50	0.50	0.50	8.9	60	1.00	0.50	0.67	56.1	142	0.75	0.75	0.75	4934	89.0	0.77	0.65	0.70	1118
	hsqldb	169.8	1	1	1	3752.8	97	1.00	1.00	1.00	36.6	55	0.00	0.00	–	532.2	443	1.00	0.67	0.80	19612	88.3	1.00	1.00	1.00	2094
	jflex	22.9	1	1	1	9.1	15	1.00	1.00	1.00	1.6	23	1.00	1.00	1.00	15.2	31	1.00	1.00	1.00	978	31.6	1.00	1.00	1.00	187
	tomcat	65.9	1	1	1	111.9	28	0.67	0.67	0.67	4.1	23	0.00	0.00	–	31.5	128	1.00	1.00	1.00	5675	128	1.00	1.00	1.00	5671
	Average	44.0	1	1	1	331	29.8	0.85	0.94	0.88	5.99	26.2	0.46	0.45	0.44	68.9	79.3	0.90	0.84	0.86	3311	67.4	0.94	0.91	0.92	988

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 https://github.com/fmselab/mixtgte.

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:

$$precision = \frac{|DET\_MFICS \cap TRUE\_MFICS|}{|DET\_MFICS|}$$

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 $DET\_MFICS \setminus TRUE\_MFICS$).

Recall is defined as:

$$recall = \frac{|DET\_MFICS \cap TRUE\_MFICS|}{|TRUE\_MFICS|}$$

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 $TRUE\_MFICS \setminus DET\_MFICS$).

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

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

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

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

[7]If a test is generated twice by a method, we count it only once.

From the results presented in Table IV, we observe that MIXTGTE 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 MIXTGTE 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 $B\bar{C}$. 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 ①, both true-mfics A and $B\bar{C}$ are contained; all the additional tests generated by SOFOT for this test (obtained by changing one parameter at a time) fail. Therefore, for test ①, SOFOT does not find any mfic, i.e., it does not find A nor $B\bar{C}$. This is due to the masking effect in test ① between A and $B\bar{C}$. The tests generated for the failing test ②, instead, correctly identifies A as mfic.

TABLE V: Execution trace of SOFOT on example SUT

Generation of CTS_2 with ACTS (to have some failing test)

#	A	B	C	*result*
1	1	1	0	failing test ①
2	1	0	1	failing test ②
3	0	1	1	pass
4	0	0	0	pass

Identification by SOFOT (tests added to find mfics)

additional tests for failing test ①

#	A	B	C	*result*
5	0	1	0	fail
6	1	0	0	fail
7	1	1	1	fail

No mfic found

additional tests for failing test ②

#	A	B	C	*result*
8	0	0	1	pass
–	1	1	1	fail
–	1	0	0	fail

A identified as mfic

BEN is the method with the lowest F-score; this is because BEN is configured to produce few 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-mfics 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-mfics of size 3, and one of size 1. The mfic of size 1 causes several tests to fail, masking the effect of the 3-way mfics. Note that, although `gcc` has four 3-way true-mfics, it takes less computation time because less tests are needed to isolate the mfics 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-mfics, 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 mfics 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 InterAIFL, FIC and SOFOT may not correctly identify multiple mfics 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 mfics 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 mfics. 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 mfics 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 (mfics), that alternates test generation and test execution. Under the assumption that the maximum strength of true-mfics is limited to t, running the process till strength t guarantees to find all and only the true-mfics; 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

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7):437–444, 1997.

[2] C. J. Colbourn and D. W. McClary. Locating and detecting arrays for interaction faults. *Journal of Combinatorial Optimization*, 15(1):17–48, Jan. 2008.

[3] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[4] A. Gargantini, J. Petke, and M. Radavelli. Combinatorial Interaction Testing for Automated Constraint Repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 239–248. IEEE, Mar. 2017.

[5] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 49–63, 2016.

[6] A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, 2018.

[7] L. S. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. BEN: A combinatorial testing-based fault localization tool. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4. IEEE, 2015.

[8] L. S. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung, and T. Xie. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, 2018.

[9] L. S. G. Ghandehari, M. N. Bourazjany, Y. Lei, R. N. Kacker, and D. R. Kuhn. Applying Combinatorial Testing to the Siemens Suite. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 362–371. IEEE, Mar. 2013.

[10] R. Jayaram and R. Krishnan. Approaches to Fault Localization in Combinatorial Testing: A Survey. In *Smart Computing and Informatics*, volume 78, pages 533–540. Springer Singapore, Singapore, 2018.

[11] D. Kuhn, R. Kacker, and Y. Lei. Practical combinatorial testing. Special publication, NIST, 2010.

[12] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[13] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[14] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating Errors Using ELAs, Covering Arrays, and Adaptive Testing Algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, Jan. 2010.

[15] C. Nie and H. Leung. The Minimal Failure-Causing Schema of Combinatorial Testing. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–38, Sept. 2011.

[16] X. Niu, N. Changhai, Y. Lei, H. K. N. Leung, and X. Wang. Identifying failure-causing schemas in the presence of multiple faults. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[17] X. Niu, N. Changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang. An interleaving approach to combinatorial testing and failure-inducing interaction identification. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[18] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Proc. of the 5th Int. Conference on Computational Science*, ICCS'05, pages 1088–1091. Springer-Verlag, 2005.

[19] M. Voelter. Using domain specific languages for product line engineering. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 329–329, Pittsburgh, PA, USA, 2009.

[20] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive Interaction Fault Location Based on Combinatorial Testing. In *2010 10th International Conference on Quality Software*, pages 495–502. IEEE, July 2010.

[21] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.

[22] J. Zhang, F. Ma, and Z. Zhang. Faulty interaction identification via constraint solving and optimization. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 186–199, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 331–341, New York, NY, USA, 2011. ACM.