A Comparison Infrastructure for Fault Characterization Algorithms

Torben Friedrichs
RWTH Aachen University
Aachen, NRW, Germany
torben.friedrichs@rwth-aachen.de

Konrad Fögen
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
foegen@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Fault characterization is an important part of combinatorial testing which enables the automatic identification of failure-inducing combinations. Up until now, many different algorithms are proposed to compute failure-inducing combinations. However, the only comparisons between different algorithms are done by the algorithms authors themselves who only evaluate few algorithms at a time which complicates comparisons. Therefore, we present a concept and a reference implementation of a comparison infrastructure that allows to evaluate fault characterization algorithms in a comparable manner. In addition, we report on the results of a preliminary comparison using the comparison infrastructure.**

***Index Terms*—Software Testing, Combinatorial Testing, Debugging, Fault Localization, Fault Characterization**

I. Introduction

Combinatorial testing (CT) is a black-box approach to reveal conformance faults between the system under test (SUT) and its specification. The research focus has traditionally been on generating test suites of minimal size that satisfy a predefined coverage criterion [1], [2]. In general, CT can only trigger failures but it cannot directly reveal faults [3]. Therefore, test inputs of failing test cases require further analysis to identify the values and value combinations that cause the failure [4]. A value or value combination that causes a test case to fail is called **failure-inducing combination** (FIC) [5]. The activity of identifying FICs is called **fault characterization** (FC) [5].

Manual FC would be a tedious and cumbersome task, especially for SUTs with many parameters and values. Therefore, a variety of algorithms for FC are proposed [6].

Fault characterization algorithms (FCA) introduce a new activity to the CT process where failing test inputs are analyzed in order to identify FICs. Furthermore, FCAs may affect the initial generation of test inputs or they may generate additional test inputs. FCAs can be classified into three categories: static, adaptive and interleaving. Each class has its own advantages and disadvantages as further elaborated in Section II.

Unfortunately, it is difficult to choose an appropriate FCA for a given test since the comparability among FCAs is limited. Each published FCA is based on different assumptions under which FICs can be identified and some publications do not define the assumptions explicitly [7]. Currently, the only comparisons between different FCAs are done by algorithms authors themselves who only evaluate very few FCAs at a time. To make matters worse, implementations of FCAs are not always available or executable [4]. Different authors also use different SUTs, fault characteristics and metrics for evaluation, e.g. different understandings of precision [8], [9].

All of this shows the need for a reliable, i.e. independent, transparent, and reproducible, comparison of FCAs and raises two research questions:

1) What are the requirements of a comparison infrastructure to allow reliable comparisons?
2) How can a comparison infrastructure be structured to enable reliable comparisons?

Therefore, we present a concept and a reference implementation of a comparison infrastructure that allows to evaluate FCAs in a comparable manner. Further, we use the comparison infrastructure for a preliminary comparison to evaluate if the presented comparison infrastructure is appropriate.

The paper is structured as follows. Section II introduces foundations of CT and FC. Section III discusses related work. In Section IV, the concept of the comparison infrastructure is presented and a reference implementation is discussed in V. The results of a preliminary comparison are shown in Section VI. Afterwards, we conclude with a summary of our work.

II. Background

A. Basic Terminology

According to IEEE [10], the concepts of error, fault and failure are distinguished as follows. An **error** is "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." A **fault** is the identified or hypothesized cause of a failure. A **failure** is an external behavior of the SUT, i.e. a behavior observable or perceivable by the user, which is incorrect with regards to the specified or expected behavior.

Combinatorial testing (CT) is a black-box testing approach based on an **input parameter model** (IPM) [1]. The IPM consists of n parameters $P = \{p_1, ..., p_n\}$. Each parameter p_i is represented as a nonempty set V_i of m_i discrete values. A **tuple** τ is a set of parameter-value pairs for d distinct parameters. Each parameter-value pair (p_i, v_j) is an assignment of value $v_j \in V_i$ for parameter p_i. A tuple that consists of $d = n$ parameter-value pairs is a **test input** to stimulate a SUT. A

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

tuple τ_a covers another tuple τ_b (denoted as $\tau_b \subseteq \tau_a$) if every parameter-value pair of τ_b also exists in τ_a.

A **combination strategy** describes how values of the IPM are selected such that a coverage criterion is satisfied [1]. A **coverage criterion** is a condition which must be satisfied by a test suite. The **t-wise coverage criterion** is a common criterion that is satisfied if each value combination of t parameters appears in at least one test input [1].

To assist a tester locating faults, test inputs of failing test cases can be further analysed to identify the values and value combinations responsible for failure [4].

A value or value combination of a test input that causes a test case to fail is a **failure-inducing combination** (FIC). In related work, this is also referred to as failure-triggering schema or failure-causing schema (e.g. [2], [5]). The activity of identifying FICs is called **fault characterization** (FC) [5].

A FIC can be described as a tuple τ_F and every test input τ that covers $\tau_F \subseteq \tau$ triggers a failure. Identifying FICs is usually not sufficient. For instance, the test input τ that fails is itself a FIC and the repeated identification is of little help. Instead, minimal FICs should be identified because they better explain the cause of a failure. A FIC τ is **minimal** (MFIC) if and only if for each proper sub-combination $\tau' \subset \tau$ there exists a test input that covers τ' and does not fail [2].

MFICs are a good starting point for further debugging and most FCAs attempt to identify MFICs. Recent work (e.g. [4], [11]) focuses on identifying MFICs in the presence of multiple faults. In that context, MFICs can mask each other making it difficult to locate the faults [11]. Therefore, Arcaini, Gargantini and Radavelli [4] introduce the notion of isolated MFICs: A MFIC τ_F is **isolated** by a test input τ if and only if τ_F is the only MFIC covered by τ [4].

For a failing test input τ, $2^n - 1$ proper sub-combinations $c \subset \tau$ exist and each sub-combination c is a FIC candidate [9]. For a FIC candidate, it must also be checked whether all test inputs that cover the candidate are indeed failing. Since this is impractical for SUTs with many parameters, *approximate solutions* are accepted to reduce the number of test inputs.

The approximate solutions are based on assumptions. Different **fault characterization algorithms** (FCA) are proposed to compute approximate solutions while the assumptions hold.

The main differences between FCAs are (1) the set of assumptions that is required to hold, (2) the quality of the results, (3) the number of test inputs required to calculate the results, and (4) the computational overhead in terms of time and memory consumption. To support the design of a comparison infrastructure, the assumptions and FCAs are further examined in the next two subsections.

B. Assumptions in Fault Characterization

Assumptions are used to limit the space that must be searched to identify a FIC. The assumptions and possible options are listed in Table I. The assumptions are not always explicitly stated when a new FCA is presented. Some Assumptions can be implicitly defined by other assumptions and assumptions can also be a prerequisite to other assumptions.

TABLE I
ASSUMPTIONS REQUIRED BY FCAS

No.	Assumptions	Options
1	Determinism of the SUT and test oracle	Yes, No
2	Distinguishable Failure Information	Yes, No
3	Multiple MFICs in the exhaustive test set	Yes, No
4	Number of parameters involved in each MFIC	$= t, \leq t, \leq n$
5	No additional MFICs during search	Yes, No
6	Existence of Safe Values	Yes, No
7	Parameters are independent	Yes, No
8	Multiple MFICs per failing test input	Yes, No
9	No coincidental correctness	Yes, No
10	Overlapping MFICS per failing test input	Yes, No
11	MFICs can be isolated	Yes, No

Assumption 1 defines whether or not the SUT and test oracle must be deterministic, i.e when executing a test input several times, the result must always be the same [11]. Most FCAs rely on a test oracle that labels each test input with either `pass` or `fail` (Assumption 2) [7]. But, some FCAs require more detailed information in order to distinguish multiple faults [9]. Assumption 3 allows or permits the exhaustive test set to only contain one MFIC. Assumption 4 defines the number of parameters that are involved in a MFIC. Assumption 5 defines whether or not additional MFICs may or may not be discovered during the identification process [9], [12]. This is also closely related to assumption 6 which may or may not require the existence of *safe values* [9]. When safe values are required, a value must exist for each parameter that is not associated with a MFIC [13]. The assumption can be further tightened by not only requiring safe values but also by requiring at least one passing test input. When the parameters are independent, i.e. no constraints exist, (assumption 7), each tuple of n safe values denotes a passing test input.

When multiple MFICs may exist in the exhaustive test set, assumption 8 states whether or not more than one MFIC may be covered by a failing test input. When multiple MFICs are allowed in one test input, coincidental correctness can distort the results [7] (assumption 9). This is an effect where two or more faults balance out and even though both faults are activated, the result is correct. In that context, some FCAs accept or do not accept overlapping MFICs (assumption 10), i.e. MFICs that share common parameter values within one test input or within the whole test suite [12]. Assumption 11 determines whether or not MFICs can be isolated [4].

C. Fault Characterization Algorithms

We conducted a literature review and identified 29 publications that present FCAs. They are listed in Table II. The FCAs are classified into three categories [14].

1) Static Fault Characterization: Static fault characterization algorithms only use the initially generated test suite to identify FICs. No further test inputs will be generated. The information of a conventionally generated test suite is often not sufficient to uniquely identify the FICS. Therefore, static FCAs either consist of a dedicated combination strategy to encode required information or they cannot give a precise solution.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

TABLE II
CLASSIFICATION OF PUBLISHED FCAS

Category	FCAs
Static	[15], [16], [17], [18], [19], [20], [13], [21], [22]
Adaptive	[23], [8], [24], [2], [7], [25], [26], [27], [28], [12], [29], [30], [31] [11], [32], [33],
Interleaving	[34], [35], [9], [4]

2) Adaptive Fault Characterization: Adaptive FCAs can be used together with conventional combination strategies. If a test suite does not contain sufficient information to identify a FIC, additional test inputs are generated by the adaptive FCA. After the execution of the new test inputs, the FCA again attempts to identify a FIC. The activities of FIC identification and test input generation are repeated until sufficient information is obtained.

3) Interleaving Fault Characterization: In adaptive FCAs, the activities of FIC identification and test input generation are separated and executed one after another. Interleaving FCAs blend both activities. One test input is generated and executed at a time. If a test input passes, the contained tuples contribute to the coverage and the next test input is generated. If a test input fails, dedicated test inputs are generated to identify the FIC. Again, each dedicated test input that passes contributes to the coverage. Once a FIC is identified, all tuples that cover the FIC are excluded from further test input generation.

It can be argued that static FCAs have the advantage of knowing upfront the number of test inputs to execute [15]. However, they cannot know which test inputs will fail. Therefore, the general encoding of information leads to larger initial test suites. If no test input fails, adaptive FCAs have no overhead compared to small test suites generated by conventional combination strategies. However in the presence of failures, additional test input executions are necessary.

Interleaving FCAs have the advantage of immediate feedback. If a test input failed, the FIC can be directly excluded from further test input generation resulting in potentially fewer test inputs and better coverage. However, the process of interleaving FCAs requires combination strategies to generate one test input at a time which potentially excludes combination strategies like `IPO` [36] and its extensions (e.g. [37], [38], [39]). In addition, it requires a permanent transition between generation, execution and fault characterization.

Beyond these general argumentation, it is mostly unclear how the three categories compare against each other. In addition, it is also unclear how different FCAs within one category compare against each other.

The results of existing comparisons are not significant because the used SUTs and implicit assumptions may affect the accuracy, number of test inputs and computational overhead.

Moreover, authors have different understandings of accuracy. For instance, `BEN` [27] is based on ranking MFIC candidates according to their *suspiciousness*. For MFICs with less than t parameters, `BEN` identifies many t-sized combinations that all cover the MFIC [14]. In contrast, MixTgTe [4] attempts to isolate MFICs such that only one MFIC is contained by a failing test input.

Different authors also use different metrics for evaluation which complicates comparisons even more. Even though precision and recall is used in most publications to measure the accuracy, the definitions can vary(e.g. [8], [9]).

All of this shows the need for an infrastructure that supports an independent, transparent and reproducible evaluation and comparison of FCAs.

III. RELATED WORK

To the best of our knowledge, there is only one survey by Jayaram and Krishnan [6] that qualitatively compares FCAs. All other comparisons are conducted by algorithm authors as listed in Table II. Of particular note is a case study conducted by Chandrasekaran et al. [40] where the `BEN` FCA [30] is applied to different software. Although, `BEN` is not compared to other FCAs in this work.

Two publications [41], [42] calculate the fault-detection probability of t-wise coverage based on identified MFICs. Their work is independent of any specific FCA.

In the context of test input generation, there are some evaluations that compare the effectiveness and efficiency of combination strategies that satisfy t-wise coverage with random testing and adaptive random testing (e.g. [43], [44]). These evaluations, while helpful when choosing combination strategies, do not include any notion of FC.

There also exist repositories like `SIR`[1] [45] providing an infrastructure for controlled experimentation and software with known faults.

One of the main problems in CT is the availability and comparability of implementations. Therefore, `CITLAB` [46], [47] is designed as a laboratory that provides a common syntax to specify IPMs. It is extensible such that new algorithms can be easily integrated and compared using the same IPMs. In recent work, `CITLAB` is also extended and migrated to a web-based implementation called `CTWEDGE` [48]. However, it only focuses on test input generation and does not include FC.

The framework `coffee4j`[2] supports the complete CT process including test input generation, execution and fault characterization. The framework provides extension points to integrate algorithms for test input generation as well as for static, adaptive and interleaving FC.

IV. COMPARISON INFRASTRUCTURE

In the following, we introduce the main concepts that the comparison infrastructure is based on. Then, we discuss the requirements that the infrastructure has to satisfy and map them to a sequence of steps that makeup the comparison process. Finally, we present a component-based architecture for the comparison infrastructure.

[1]Software Infrastructure Repository (SIR): https://sir.csc.ncsu.edu/
[2]coffee4j: https://coffee4j.github.io

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

A. Concepts

The comparison infrastructure is based on three main concepts: FCA (already defined in Section II), metric, and evaluation scenario. . A **metric** is "a measure in terms of an attribute" and an attribute is a "[...] characteristic of an entity that can be distinguished[...]" [49].

The execution of FCAs requires an IPM that defines the input space and coverage criterion, a SUT which is stimulated by test inputs and a test oracle that labels test inputs with either `pass` or `fail`. To make informed statements, the faults of the SUT must also be known upfront. Therefore, an **evaluation scenario** consists of an IPM, a SUT and a test oracle.

A **comparison** is the process of applying a set of FCAs to a set of evaluation scenarios and comparing their characteristics by computing a set of metrics.

B. Requirements

Overall, the comparison infrastructure shall facilitate reliable comparisons between FCAs. This goal can be broken down into the following major requirements:

Scenario Modeling: The infrastructure shall allow to specify evaluation scenarios.

FCA Execution: The infrastructure shall execute FCAs on evaluation scenarios. New FCAs shall be used without making assumptions about the implementation language.

Reliable Comparison: The infrastructure shall provide the means for reliable, i.e. reproducible, independent and transparent, FCA comparisons. This means that the results must not be influenced by the execution order or other external factors.

Scenario Analysis: The infrastructure shall analyze the difficulty of evaluation scenarios based on values like number of parameters, number of MFICs or satisfied or violated FCA assumptions in order to rate FCAs accordingly.

FCA Analysis: The infrastructure shall analyze and classify information gathered during the FCA execution according to predefined evaluation metrics. These metrics shall define a standard to measure the performance of an FCA regarding run-time and quality of the returned MFICs.

Visualization: The infrastructure shall visualize the analysis results to provide insights to further refine the comparisons or to make informed decisions about FCA application.

C. Comparison Process

Based on the requirements, we propose the following process, depicted in Fig. 1, to compare FCAs.

At first, all evaluation scenarios are defined. Then, the infrastructure selects an FCA and an evaluation scenario. It executes the FCA on the evaluation scenario to create a **trace**. A trace stores the execution time, all test inputs, and the indicated MFICs. This means that all non-interleaved FCAs need to include a CT algorithm as well.

After the execution, the infrastructure analyzes the evaluation scenario and trace by computing predefined metrics.

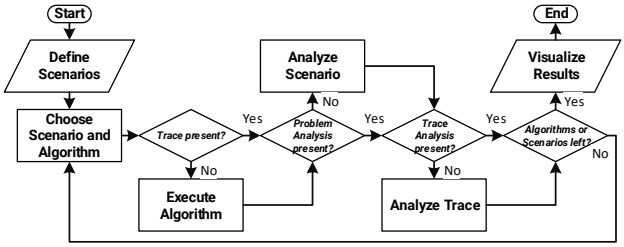

Fig. 1. Overview of the Comparison Process

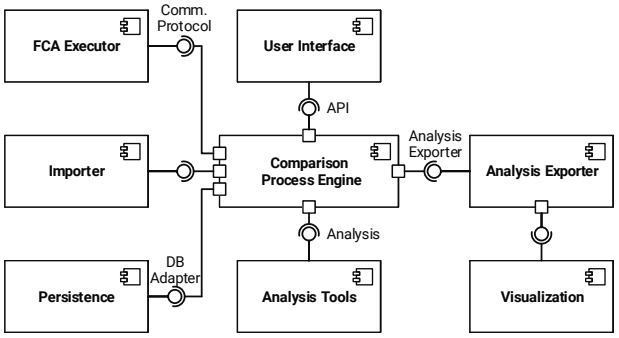

Fig. 2. Top-Level Architecture of the Comparison Infrastructure

This process is repeated until no combination of FCA and evaluation scenario remains. Finally, the results are visualized.

Additionally, each step checks whether a result already exists. If it does, the corresponding step is skipped which allows interruption and continuation of the process.

D. Component-based Architecture

To realize the proposed comparison process, we designed a component-based software architecture shown in Fig. 2.

The **FCA Executor** component provides the interface to execute FCAs. As all communication between the infrastructure and an FCA is done over a standard communication protocol, the infrastructure is strongly decoupled from the used FCAs.

The **Importer** component deals with the transformation of evaluation scenarios into an internal representation. Furthermore, it provides means to add new input formats to integrate already existing scenario descriptions.

The **Persistence** component stores and retrieves of all information produced by the infrastructure.

The **Analysis Tools** component provides all metrics used in the FCA analysis as well as all assumption checks used in the evaluation scenario analysis. Any assumption or metric not already implemented can be easily added.

The **Analysis Exporter** component allows to export all produced analysis results to further explore the data with other tools or provide their own custom visualizations.

The **Visualization** component provides a default visualization of the comparison results. It is independent of the infrastructure itself and obtains the necessary data using the analysis exporter.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

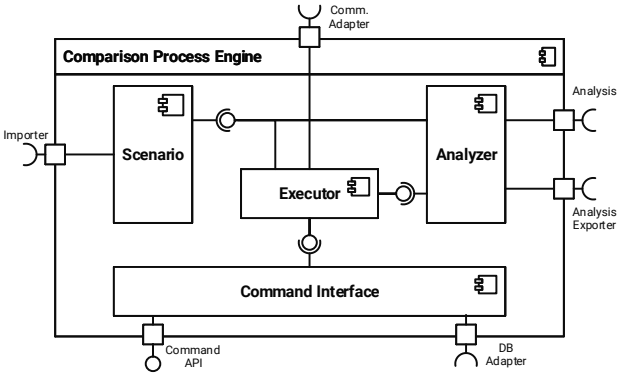

Fig. 3. Architecture of the Comparison Process Engine

```
Model Printer

Parameters:
Side:   {Single Double}
Color:  {Colored Gray B&W}
Format: {A2 A4}
Scale:  {50% 100% 200%}

Constraints:
C1: # Scale=200% => Color!=Colored  #
C2: # Side=Single || Scale!=50%     #
C3: # !(Format=A2 && Side=Double)   #

Evaluation Scenarios:
S1: # 2 (C3) (C1 C2) #
S2: # 3 (C2 C3) (C1) #
```

Listing 1. Extended CTWEDGE Model

The **User Interface** component allows users to specify which algorithms and scenarios to compare, to configure different execution parameters, and to generally control the comparison process.

The **Comparison Process Engine** component implements and orchestrates the defined comparison process. It contains four sub-components, depicted in Fig. 3. The **Scenario** component stores the internal representations of the evaluation scenarios and validates their correctness. The **Executor** component initializes the FCAs with the correct data, executes them and records their traces. The **Analyzer** applies the predefined metrics to the given evaluations scenarios or traces.

The Comparison Process Engine is controlled via a **Command Interface** component, which delegates given commands to components and retrieves the necessary data.

V. CAFFEINE – A Reference Implementation

In the following, we present a reference implementation of the comparison infrastructure called CAFFEINE, the **C**omp**a**rison In**f**rastructure for **F**ault Charact**e**ri**z**ation**e**. It implements the comparison process and realizes the component-based architecture. CAFFEINE offers a command line interface that allows to specify evaluation scenarios, provide FCA implementations, and control the comparison process. The source code is publicly available[3].

A. Input Format

The default format to specify evaluation scenarios is based on the input language of CTWEDGE. Because it is designed only for CT modeling, we extended the language to also support the modeling of named constraints and evaluation scenarios. Listing 1 shows an example.

Each evaluation scenario has a unique identifier (*Printer* in the example) in order that the comparison results can be matched back later.

First, the IPM is defined. Each parameter has an identifier and a domain. Domains for a parameter `P` can be defined as Boolean (`P: Boolean`), sets (`P: {a, b, c}`), or ranges (`P = [1...2]`).

[3]https://github.com/coffee4j/caffeine

Next, so called named constraints (analogous to named parameters) are specified. They are delimited by # tokens and can be defined in predicate logic with relations between parameters and their values. A complete definition of the constraint language can be found in [48]. Any potential MFICs must also be modeled here.

Finally, evaluation scenarios are defined. Each scenario has a unique identifier and is surrounded by # tokens. It contains three parts: (1) the testing strength t, (2) a list of all constraints that model faults, (3) a list of constraints that model dependencies between parameters. Because each constraint can potentially describe a large number of combinations, the number of MFIC entries in this list can be much smaller than the actual number of MFICs present in a scenario.

B. FCA Execution

The way FCAs are executed and measured is of vital importance to the comparison infrastructure. Because the infrastructure can run algorithms written in any language and framework, each FCA is executed in a separate process communicating with the infrastructure over a communication protocol. All FCAs selected for a comparison are specified in a configuration file by means of key-value pairs that map FCA names to executable commands. An example selecting two FCAs is shown in Listing 2.

```
{
  "Aifl": "java -jar c4j.jar -g Ipog -a Aifl",
  "Ofot": "python ofot-adapter.py"
}
```

Listing 2. Algorithm Configuration File

Each FCA Executor implements a communication protocol to exchange messages during the comparison process. The communication has two phases: *Initialization* and *Execution*.

For initialization, the infrastructure at first transmits the testing strength and the IPM. Then it sends any known forbidden combinations prefixed by FORBIDDEN. Unset parameter values are substituted by `-1`. The end of this phase is signalled by sending the token START. An example for the evaluation scenario S1 of Listing 1 is given in Listing 3.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

```
2
2 3 2 3
FORBIDDEN -1 0 -1 2
FORBIDDEN 1 -1 -1 0
START
```

Listing 3. Example Messages during Initialization Phase

During the execution, control is turned over to the FCA. There are three different message types that the algorithm can send: (1) generated test inputs prefixed by a `?`, (2) found MFICs prefixed by `!`, and (3) assumptions prefixed by `#` followed by the assumption identifier and either the keywords `SATISFIED` or `VIOLATED`.

Upon receiving a test input, the infrastructure immediately checks the test input against the scenario and answers with either `SUCCESS` or `FAIL`. Additionally, the name of the constraint or MFIC that caused a failure is added if applicable. All messages are saved by the infrastructure in a respective trace which can later be analyzed.

```
? 0 0 0 0       |   SUCCESS
? 0 0 0 2       |   FAIL C1
? 0 1 0 2       |   FAIL C3
# A1 VIOLATED   |
? 0 0 0 1       |   SUCCESS
! -1 -1 0 2     |
```

Listing 4. Example Messages during Execution Phase

An example with four test inputs is shown in Listing 4. In two cases, the tests fail, due to the constraints C1 and C3. During the execution, the FCA additionally reported that the assumption A1 has been violated. Finally, the algorithm returns the MFIC `(-1, -1, 0, 2)` where the values of the first two parameters do not matter.

C. Analysis

The **Analzer** implements two main functions: (1) The evaluation scenario analysis determines the difficulty of the evaluation scenario to evaluate FCAs according to their intended application. If, for example, a SUT is simple, then it makes no sense to use an algorithm that trades performance in easy cases for better performance on harder cases. (2) The FCA analysis evaluates the performance of FCAs.

In `CAFFEINE`, both analysis functions are realized using the same mechanism. Therefore, the **Analysis Tools** component provides three interfaces to provide analysis metrics: (1) **IPM Analysis** for metrics that only require data stored in the IPM, e.g. basic metrics like the number of parameters, (2) **Scenario Analysis** for metrics needing evaluation scenario information, e.g. metrics like number of valid combinations or number and size of the contained faults, and (3) **Trace Analysis** for metrics relying on the FCA performance and any assumptions reported during the execution.

Given implementations of these interfaces can be registered to an **Analyzer** which calls them when applicable. The infrastructure only needs to compute each **IPM Analysis** and **Scenario Analysis** once because the results stay valid for all algorithms while **Trace Analysis** has to be run multiple times.

TABLE III
DEDICATED EVALUATION SCENARIOS

Name	Parameters	Forbidden Combinations
DES-1	$2^5 4^2$	5
DES-2	$2^6 4^2$	5
DES-3	$5^1 2^4 4^2$	4
DES-4	$2^2 4^2$	4
DES-5	$2^9 4^4$	6
DES-6	$2^6 4^1 10^1$	5
DES-7	2^{20}	4
DES-8	4^4	7
DES-9	$2^1 3^2 4^1$	5
DES-10	$2^2 3^1 4^2$	8

D. Visualization

The **Visualization** component provides means for quick and interactive data exploration. A screenshot of the main interface is displayed in Fig. 4.

It is divided into three parts. In the top-left a table shows the number of completed, timed-out, and invalid iterations for each algorithm. Below are various sliders that allow users to filter the data. The main part shows a bar chart where analysis metrics can be displayed next to each other. This is achieved by selecting one or more metrics from a multi-select dropdown. This area also allows the user to switch to a line chart where two metrics can be plotted against each other.

Because the exported data may include a variety of different metrics, the visualization can be configured accordingly.

VI. PRELIMINARY COMPARISON OF FAULT CHARACTERIZATION ALGORITHMS

As a proof-of-concept to show the applicability of the proposed comparison infrastructure and its reference implementation `CAFFEINE`, we present the results of a preliminary FCA comparison.

A. Evaluation Scenarios

One important decision to be taken when evaluating software testing techniques is the choice of SUTs.

Therefore, we created two sets of SUTs to define the evaluation scenarios used in the comparison: one set of dedicated SUTs (created by us) and another set of SUTs taken from the `CITLAB` repository[4] which is commonly used in CT research.

Dedicated SUTs offer the chance to create evaluation scenarios that have similar characteristics as real SUTs, while being small enough for a reasonably quick comparison time. One notable property of most evaluation scenarios is that the majority of input parameters are of type Boolean to model typical real-life configurations options that are either-or choices. An overview of the used dedicated evaluation scenarios is given in Table III.

In addition, 10 evaluation scenarios with constraints are taken from the `CITLAB` repository. They are shown in Table IV. In general, these evaluation scenarios are much larger

[4] `CITLAB` Repository: https://sourceforge.net/p/citlab/wiki/Benchmarks/

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

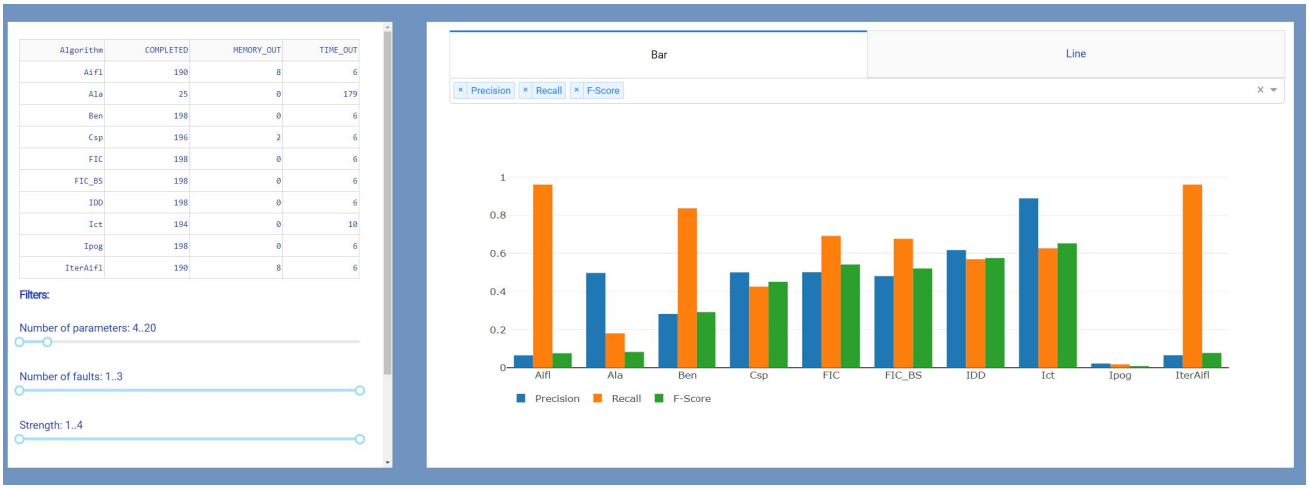

Fig. 4. Dashboard Visualizing Comparison Results

TABLE IV
CITLAB EVALUATION SCENARIOS

Name	Parameters	Forbidden Combinations
Banking1	$3^4 4^1$	111
Banking2	$2^{14} 4^1$	3
bugzilla	$2^{48} 3^1 4^2$	5
CommProtocol	$2^{10} 7^1$	127
Concurrency	2^5	7
gcc	$2^{188} 3^{10}$	40
Healthcare1	$2^6 3^2 5^1 6^1$	21
Healthcare2	$2^5 3^6 4^1$	24
HeartBleed	$2^1 16^4$	65535
SmartHome	2^{37}	158

and more heavily constrained. Because these evaluation scenarios are meant for the evaluation of CT, they do not contain any fault information.

For both sets, two faults of strength 2, two faults of strength 3 and two faults of strength 4 are randomly distributed over all parameters. These faults are then combined into evaluation scenarios with up to three of them being active at a time. The testing strength for each evaluation scenario is set to match the highest strength of any contained fault. Moreover, the randomly generated faults are checked for validity to avoid faults that are excluded by constraints and at the same time are completely covered by another fault. In total, six evaluation scenarios per comparison are used. The comparison is run once with all constraints deactivated, and once more with all constraints activated. This allows to compare how the performance of FCAs is influenced by constraints.

All comparisons are run on a single computer running Fedora 30 with an Intel i5-3570k@4x3.80 GHz processor. Each FCA was assigned eight GB of memory and run for three iterations per evaluation scenario with a timeout of 15 minutes per iteration. All FCAs are implemented in `coffee4j`.

B. Applied Evaluation Metrics

We use four metrics to assess and compare FCAs.

1) Test Inputs: The number of test inputs is the primary metric to evaluate the efficiency of a FCA. Each algorithm makes a trade-off between the number of test inputs and the quality of the results.

2) Execution Time: The execution time delivers a direct approximation for the effort spent by the FCA. However, this metric may be unreliable. For instance, because execution times can fluctuate depending on additional activities on the host system. This can be counteracted by running multiple iterations, but cannot be completely eliminated.

Nonetheless, the execution time is still a very useful metric to evaluate the effort spent by an FCA because this is a main limiting factor for applying FCAs in practice.

3) Naïve FCA Quality (N-FCA-Q): The easiest way to determine the quality of FCAs is to compare the found FICs with the FICs specified in the evaluation scenario.

This metric computes three values: Precision, Recall, and F-Score. *Precision* describes the percentage of FICs found by the FCA that are real FICs of the evaluation scenario. It is a notion of *correctness* because the precision value is lowered by every found FIC that is not specified in the evaluation scenario. It is calculated as $precision = \frac{|found \cap real|}{|found|}$.

Technically, Precision is undefined when no FIC is found, i.e. $|found| = 0$. This is an edge case to which we assign a precision of 1 because no incorrect FIC is found.

Recall describes the percentage of real FICs that are found by a FCA. In that sense, it is a notion of *completeness* which is calculated as $recall = \frac{|found \cap real|}{|real|}$. Technically, Recall is not defined when an evaluation scenario does not contain FICs, i.e. $|real| = 0$. This is an edge case that can be ignored because FC is not needed without FICs.

Precision and Recall should not be interpreted in isolation because both values can be easily optimized. Bad FCAs that never find any FICs would achieve a precision of 1. Although, the corresponding recall would be 0. Also, bad FCAs could simply return all possible value combinations of an evaluation

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

scenario to achieve a recall of 1. However, the corresponding precision would be very low.

Instead, good FCAs should have high precision and high recall at the same time. Therefore, *F-Score* describes the harmonic mean between Precision and Recall. It is calculated as *F-Score* $=2 \cdot \frac{\text{precision} \cdot recall}{\text{precision}+\text{recall}}$.

However, this naïve approach to evaluate the quality of FCA results has a disadvantage, because very often FCAs are not able to find the exact FICs, but they rather find approximations, i.e. larger FICs that cover the actual FICs. This metric would evaluate all of these FCAs the same, no matter how good the approximations might be.

4) Classification-based FCA Quality (C-FCA-Q): This metric considers the quality of FCAs as a classification problem to overcome the disadvantage of the naïve approach. The quality of a FCA and its found FICs is determined on the level of test inputs. Therefore, the exhaustive set of test inputs as defined by the evaluation scenario is analyzed.

A test input is a *true positive* (tp) if it covers a FIC found by the FCA and if it also covers a real FIC specified by the evaluation scenario.

A test input is a *true negative* (tn) if it does not cover any FIC found by the FCA and if it also does not cover any real FIC.

A test input is *false positive* (fp) if it covers a FIC found by the FCA but if it does not cover a real FIC.

A test input is a *false negative* (fn) if it does not cover any FIC found by the FCA but if it covers a real FIC.

Similarly to the N-FCA-Q metric Precision, Recall and F-Score can be calculated by *precision* $=\frac{tp}{tp+fp}$, *recall* $=\frac{tp}{tp+fn}$, and *F-Score* $=2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$.

C. Compared Algorithms

Our preliminary comparison comprises the following FCAs: `Aifl` [23], `IterAifl` [8], `BEN` [27], `IDD` [25], `CSP` [20], `FIC` [7], `FIC_BS` [7], and `ICT` [9]. By that selection, all three FCA categories are covered as well as FCAs that support and ignore constraints. In addition, `IPOG` [37] is considered as a very basic static FCA and is used as a baseline for comparisons.

D. Results & Discussion

An observation we made was that not all FCAs successfully completed all iterations. Notable are the FCAs `Aifl` and `IterAifl` which run out of memory in 4 of the dedicated and 9 of the `CITLAB` evaluation scenarios. To obtain comparable results, only evaluation scenarios that are completed by all FCAs are analyzed.

Table V shows the average precision, recall and F-Score values of the N-FCA-Q and C-FCA-Q metrics for each FCA. Most notably, the FCA quality decreases when introducing constraints, especially when using the N-FCA-Q metric. The difference to the C-FCA-Q metric is partially caused by the fact that the C-FCA-Q metric is only calculated for dedicated evaluation scenarios due to time constraints.

Overall, `ICT` is clearly the best FCA concerning the N-FCA-Q metric, while `IterAifl` is the best algorithm regarding C-FCA-Q. Nonetheless, all algorithms show a remarkable improvement compared to `IPOG`.

TABLE V
AVERAGE OF N-FCA-Q AND C-FCA-Q VALUES IN % (UNCONSTRAINED|CONSTRAINED)

	N-FCA-Q			C-FCA-Q		
FCA	**pre.**	**rec.**	**f-s.**	**pre.**	**rec.**	**f-s.**
Aifl	7|7	100|88	10|6	92|73	100|94	95|76
Ben	41|25	100|62	43|24	78|69	100|91	85|71
Csp	52|49	46|39	48|42	74|73	87|81	77|74
FIC	73|29	88|43	78|29	76|46	100|96	84|54
FIC_BS	70|26	86|40	75|26	67|42	100|96	76|50
IDD	79|47	75|38	76|40	90|69	99|91	93|72
Ict	97|78	83|50	85|53	100|90	90|77	90|72
IterAifl	7|7	100|88	10|6	100|83	100|94	100|83
Ipog	3|4	6|0	3|0	100|100	25|17	32|24

TABLE VI
OVERHEAD RELATING TO IPOG IN % (UNCONSTRAINED|CONSTRAINED)

FCA	Overhead Test Inputs	Overhead Execution Time
Aifl	304 | 292	129 | 114
Ben	150 | 153	156 | 119
Csp	100 | 100	308 | 180
FIC	341 | 359	105 | 103
FIC_BS	248 | 216	104 | 103
IDD	120 | 117	103 | 101
Ict	151 | 130	888 | 441
IterAifl	3009 | 5556	379 | 252
Ipog	100 | 100	100 | 100

Table VI shows the overhead of each FCA in regard to the number of test inputs and execution time. As can be seen, the execution times can be divided into three categories: (1) FCAs with almost no overhead like `IDD` or `FIC`, (2) FCAs with a medium overhead taking about twice as long as CT only like `CSP` or `IterAifl`, and (3) the FCA `ICT` which needs almost nine times as much time. This is partially caused by the algorithm itself due to the feedback checking mechanism, but mostly influenced by the need for a different, less efficient CT algorithm.

This extraordinary long execution time, however, is put into context when looking at the number of additional test inputs. Here `ICT` places third behind `CSP`, a static FCA, and `IDD`. The outlier in this category is `IterAifl` needing between 30 and 50 times as many test inputs as CT. Another interesting observation can be made when looking at the relationship between the number of test inputs and the number of MFICs in the evaluation scenario, depicted in Fig. 5.

It shows that `ICT` has an overhead smaller than one for three MFICs. This is caused by evaluation scenarios where the FCA finds an MFIC with a strength smaller than the testing strength. This allows `ICT` to quickly eliminate larger portions of the search space. Overall, this means that in some cases the application of an interleaved FCA is more efficient concerning the number of test inputs than CT alone.

E. Threats to Validity

While the obtained results of this initial FCA comparison paint a pretty clear picture, there are threats that can

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

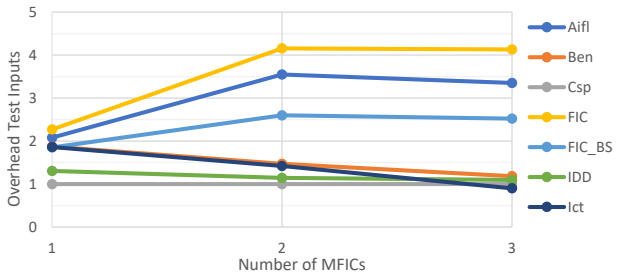

Fig. 5. Relationship between Number of MFICs and Test Input Overhead

lead to misleading interpretation. First, the choice of SUTs and evaluations scenarios. While both the dedicated and the CITLAB evaluation scenarios provide abstractions from real-life systems, they are not representative for all software. This is especially true for the contained faults which are randomly distributed in the SUTs.

Second, FCAs are often only published without an access to a respective implementation of the algorithms that could be used for a comparison. This requires a re-implementation of algorithms which can result in a slower or incorrect implementation compared to the original one. This is especially risky because sometime not all the details of an algorithm are described in the publication.

Third, there are many different approaches and metrics to measure the quality of FCAs. For the performed comparison an initial set of metrics is used. Other metrics should be defined and compared to the ones proposed by us.

Fourth, this comparison does not include all published FCAs. Therefore, the obtained results cannot be generalized.

To overcome this threats, the comparison infrastructure is designed to be extensible such that different evaluation scenarios, other FCA implementations as well as different metrics can be integrated. To allow and support the reliability of the experiment, the comparison infrastructure itself, the FCA implementations as well as the used evaluation scenarios and metrics are publicly available[3].

VII. Conclusion

CT can only trigger failures but cannot directly reveal faults. To locate a fault, further debugging activities are required. Test inputs of failing test cases can be analysed to identify values and value combinations that induce failures. Unfortunately, exact solutions are often impractical for SUTs with many parameters. For a failing test case τ, $2^n - 1$ proper sub-combinations $c \subset \tau$ exist and each proper sub-combination can be failure-inducing. Therefore, approximate solutions are accepted where the search space is reduced by requiring that certain assumptions hold.

Up until now, a variety of different FCAs that compute approximate solutions are published. They differ in terms of the set of assumptions that is required to hold, the quality of the results as well as the number of required test inputs and the computational overhead. Unfortunately, it is difficult to choose an appropriate FCA for a given test because a comprehensive comparison is lacking.

As a first step towards a comprehensive comparison of FCAs, we present the concept of a comparison infrastructure that allows an independent, reproducible and extensible comparison of FCAs. Therefore, requirements concerning the infrastructure are defined first and a comparison process is proposed. In addition, key components are identified and a top-level architecture is designed.

Furthermore, we present CAFFEINE, a reference implementation of the comparison infrastructure . All functionality is provided through a command line interface with three top-level commands: model, trace, and analysis.

To demonstrate the applicability of the reference implementation, a preliminary comparison of FCAs is also conducted.

The results show that all FCAs provide much better identification of MFICs while having a manageable overhead. A special case is presented by which the interleaving approach, in some situations, is even more efficient concerning the number of test inputs than CT alone.

As future work, we will extend the preliminary comparison to cover all major FCAs and we will systematically design test scenarios according to the assumptions of the FCAs.

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Softw. Test., Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.
[2] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 15:1–15:38, Sep. 2011.
[3] A. Arcuri and L. Briand, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1088–1099, Sep. 2012.
[4] P. Arcaini, A. Gargantini, and M. Radavelli, "Efficient and guaranteed detection of t-way failure-inducing combinations," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2019, pp. 200–209.
[5] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, 2011.
[6] R. Jayaram and R. Krishnan, "Approaches to fault localization in combinatorial testing: A survey," in *Smart Computing and Informatics*, S. C. Satapathy, V. Bhateja, and S. Das, Eds. Singapore: Springer Singapore, 2018, pp. 533–540.
[7] 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*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 331–341.
[8] 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*, July 2010, pp. 495–502.
[9] 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*, pp. 1–1, 2018.
[10] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610.12-1990, Dec 1990.
[11] 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*, pp. 1–1, 2018.
[12] X. Niu, C. Nie, Y. Lei, and A. T. S. Chan, "Identifying failure-inducing combinations using tuple relationship," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, March 2013, pp. 271–280.
[13] W. Zheng, X. Wu, D. Hu, and Q. Zhu, "Locating minimal fault interaction in combinatorial testing," *Adv. Software Engineering*, vol. 2016, pp. 2 409 521:1–2 409 521:10, 2016.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042

[14] J. Bonn, K. Fögen, and H. Lichter, "A framework for automated combinatorial test generation, execution, and fault characterization," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, 2019, pp. 224–233.

[15] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, Jan 2006.

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

[17] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Algorithms to locate errors using covering arrays," in *LATIN 2008: Theoretical Informatics, 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008, Proceedings*, 2008, pp. 504–519.

[18] ——, "Locating errors using elas, covering arrays, and adaptive testing algorithms," *SIAM J. Discrete Math.*, vol. 23, no. 4, pp. 1776–1799, 2009.

[19] S. Fouché, M. B. Cohen, and A. A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 177–188.

[20] J. Zhang, F. Ma, and Z. Zhang, "Faulty interaction identification via constraint solving and optimization," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, pp. 186–199.

[21] K. Nishiura, E. Choi, and O. Mizuno, "Improving faulty interaction localization using logistic regression," in *2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017*, 2017, pp. 138–149.

[22] T. Konishi, H. Kojima, H. Nakagawa, and T. Tsuchiya, "Finding minimum locating arrays using a SAT solver," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 276–277.

[23] L. Shi, C. Nie, and B. Xu, "A software debugging method based on pairwise testing," in *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA, USA, May 22-25, 2005, Proceedings, Part III*, 2005, pp. 1088–1091.

[24] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. A. Porter, "Feedback driven adaptive combinatorial testing," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 243–253.

[25] J. Li, C. Nie, and Y. Lei, "Improved delta debugging based on combinatorial testing," in *2012 12th International Conference on Quality Software*, Aug 2012, pp. 102–105.

[26] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and D. R. Kuhn, "Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 620–623.

[27] L. S. G. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn, and R. Kacker, "Identifying failure-inducing combinations in a combinatorial test set," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 370–379.

[28] L. S. G. Ghandehari, Y. Lei, D. C. Kung, R. Kacker, and D. R. Kuhn, "Fault localization based on failure-inducing combinations," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, 2013, pp. 168–177.

[29] T. Nagamoto, H. Kojima, H. Nakagawa, and T. Tsuchiya, "Locating a faulty interaction in pair-wise testing," in *20th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2014, Singapore, November 18-21, 2014*, 2014, pp. 155–156.

[30] L. S. G. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn, "BEN: A combinatorial testing-based fault localization tool," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–4.

[31] Y. Qi, Q. Wang, C. Xu, T. He, and Z. Wang, "An efficient algorithm to identify minimal failure-causing schemas from exhaustive test suite," in *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1-3, 2016.*, 2016, pp. 655–656.

[32] 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*, pp. 1–1, 2018.

[33] D. Blue, A. Hicks, R. Rawlins, and R. Tzoref-Brill, "Practical fault localization with combinatorial test design," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2019, pp. 268–271.

[34] C. Nie, H. Leung, and K. Cai, "Adaptive combinatorial testing," in *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*, 2013, pp. 284–287.

[35] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. A. Porter, "Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 43–66, 2014.

[36] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*, 1998, pp. 254–261.

[37] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA*, 2007, pp. 549–556.

[38] K. Kleine and D. E. Simos, "An efficient design and implementation of the in-parameter-order algorithm," *Mathematics in Computer Science*, vol. 12, no. 1, pp. 51–67, 2018.

[39] K. Fögen and H. Lichter, "Combinatorial robustness testing with negative test cases," in *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*, 2019, pp. 34–45.

[40] J. Chandrasekaran, L. S. Ghandehari, Y. Lei, R. Kacker, and D. R. Kuhn, "Evaluating the effectiveness of ben in localizing different types of software fault," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2016, pp. 26–34.

[41] Z. Wang and Y. Qi, "Why combinatorial testing works: Analyzing minimal failure-causing schemas in logic expressions," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–5.

[42] C. Xu, Y. Qi, Z. Wang, and W. Zhang, "Analyzing minimal failure-causing schemas in siemens suite," in *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, 2016, pp. 35–38.

[43] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Trans. Software Eng.*, vol. 41, no. 9, pp. 901–924, 2015.

[44] H. Wu, n. changhai, J. Petke, Y. Jia, and M. Harman, "An empirical comparison of combinatorial testing, random testing and adaptive random testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[45] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[46] A. Gargantini and P. Vavassori, "CITLAB: A laboratory for combinatorial interaction testing," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 559–568.

[47] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial interaction testing with CITLAB," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 376–382.

[48] 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, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, 2018, pp. 308–317.

[49] ISO/IEC 15939:2007, "Software Engineering - Software Measurement Process," 2007.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 201-210
DOI 10.1109/ICSTW50294.2020.00042