2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

A Framework for Automated Combinatorial Test Generation, Execution, and Fault Characterization

Joshua Bonn
RWTH Aachen University
Aachen, NRW, Germany
joshua.bonn@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, enabling it to automatically narrow down failed test inputs to specific failure-inducing combinations. As most current fault characterization algorithms adaptively generate more test inputs based on previous test execution results, a framework that integrates modelling, generation, execution, and fault characterization is necessary. Until now, no such framework exists, resulting in much manual work needed to identify failure-inducing combinations. We therefore introduce COFFEe, which is a framework for completely automatic combinatorial testing and fault characterization. In this paper, we derive an architecture for the framework and present coffee4j, a Java implementation of COFFEe that integrates the JUnit5 test framework.**

***Keywords*-Software Testing, Test Generation, Combinatorial Testing, Fault Localization, Fault Characterization**

I. Introduction

Combinatorial testing (CT) is a well-known black-box approach to systematically detect errors in a system under test (SUT) based on its specification. Empirical research indicates that exhaustive testing is neither feasible nor necessary. Instead, testing all pairwise value combinations should trigger most failures, and testing all 4- to 6-wise value combinations should detect all of them [1], [2]. For instance, an example by Kuhn and Wallace [1] describes that "a microwave oven control module fails when power is set on `High` and time is set to `20` minutes". In other words, the interaction of `power=High` and `time=20` triggers the failure. It is called a *failure-inducing combination* (FIC). To trigger the failure, testing all pairwise combinations is as effective as exhaustive testing, even though the pairwise test suite is much smaller.

The generation of effective test suites is not the only challenge in CT. After test execution, the test inputs of failing tests require further analysis to reveal the FICs. Manual analysis is a tedious and cumbersome task, especially for SUTs with many input parameters and values. Therefore, algorithms to automatically identify FICs are proposed. They extend CT to not only cover the generation of initial test suites but also the analysis of test execution results [3]. Nie and Leung [4] refer to this activity as *fault characterization* (FC).

The information which can be extracted from failing and succeeding test inputs is often not enough to identify a FIC. Then, adaptive algorithms generate additional test inputs, and their execution results help to further isolate the FIC. If the information is still not sufficient, the generation of additional test inputs, their execution, and the analysis of execution results continues.

Even though CT and FC algorithms are able to automatically generate test inputs and identify FICs, there exists no approach which integrates both activities. In a typical process consisting of CT and FC, one tool generates the initial test inputs, another tool or a tester applies them to test the SUT, and afterwards the execution results are passed to yet another tool for fault characterization. Then, either a FIC is identified or additional test inputs are generated, which must be executed again. Since each activity requires the tester to copy and transform information between test generation, test execution, and fault characterization tools, the process becomes tedious and error-prone.

Therefore, it would be desirable to have an integration of all activities, namely input parameter modelling, test generation, test execution, and fault characterization. Then, the tester only models the input parameters and develops executable test scripts, which describe the needed test steps to be performed. A tool could automatically handle the generation of test inputs, execution of the test scripts with generated inputs, and fault characterization, i.e. the computation of FICs and, if needed, the generation of additional test inputs.

In this paper, we therefore present a concept and a framework architecture, which integrates modelling, generation, execution, and fault characterization. Furthermore, the framework provides extension points for the exchange and use of different CT and FC algorithms. We also present an implementation of the framework and its integration with JUnit5 [5] — one of today's most popular test automation frameworks.

The paper is structured as as follows. Section II introduces fundamentals of CT and FC, and Section III discusses related work. The concept and architecture of the framework are explained in Section IV. First, a search for existing algorithms and their information needs is conducted. Based on the findings, the interfaces and overall architecture are designed. To realize the concept and architecture, we then present a Java implementation and integration with JUnit5 in Section V. Section VI demonstrates the applicability of the framework by executing it with different algorithms, SUTs, and injected faults. We conclude with a summary in Section VII.

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00057

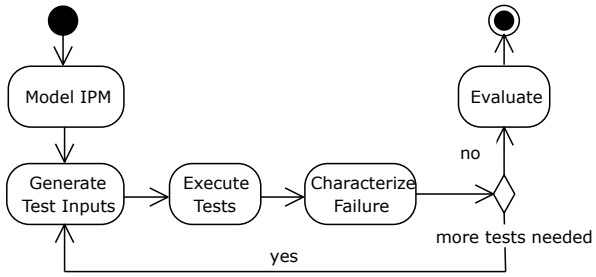

Figure 1. CT Process

Name	Values				
OS	Windows	Linux	MacOS	Android	iOS
Browser	Chrome	Edge	Firefox	Safari	
Ping	10 ms	100 ms	1000 ms		
Speed	1 KB/s	10 KB/s	100 KB/s	1000 KB/s	

Table I
AN EXEMPLARY IPM

II. BACKGROUND

Combinatorial testing (CT) is a black-box testing technique used to reveal interaction failures. Varying test inputs are generated, which are then used to exercise the SUT. Nie and Leung [4] divide CT into a process consisting of 5 individual activities. Figure 1 depicts a tailored version. The process starts with the definition of an input parameter model of the SUT capturing varying input values. Then, the generation activity uses this model as an input to generate a set of test inputs (test suite). Each test input is then used to test the SUT in order to obtain a result of either failure or success. The subsequent activity of fault characterization either identifies values that are responsible for failure or generates new test inputs, which must be tested to further isolate values responsible for failure. The process ends with a data collection and evaluation activity.

In the following, the first four process activities are further described.

A. Combinatorial Test Modelling

In CT, relevant inputs are represented as input parameters. The collection of all n parameters is called the *input parameter model* (IPM). Each parameter has a number of possible values [6]. For example, consider the IPM for an online browser game depicted in Table I. There are four parameters `OS`, `Browser`, `Ping`, and `Speed`, and each has an arbitrary number of values, e.g. `Windows`, `Linux`, `MacOS`, `Android`, and `iOS` are the values of parameter `OS`.

The activity of modelling is a creative task, which is manually performed by a tester who has to identify *suitable* input parameters and values [6].

B. Combinatorial Test Generation

To generate a set of test inputs (*test suite*), a combination strategy is applied to an IPM [7]. A combination strategy defines how values of the input parameters are selected and how they are composed to test inputs. Usually, a coverage criterion is associated to a combination strategy such that the generated test suite adheres to the coverage criterion if the combination strategy is correctly applied [7].

For instance, the t-wise coverage criterion requires the test suite to contain each combination of values between any t parameters at least once [7]. Consequently, if we look at the example of Table I with pairwise coverage ($t = 2$), each value combination of the parameters `OS` and `Browser` has to appear in at least one test input.

Since each test input contains multiple sub-combinations of size t with $t < n$, combinatorial testing strategies lead to fewer test inputs than exhaustive testing, while still maintaining a high fault detection rate [1], [2].

Combination strategies are usually automated. IPOG [8] is a well-known strategy, which can generate t-wise test suites. It iteratively extends an initial set of incomplete test inputs until it encompasses all parameters defined in the IPM with the given strength.

C. Combinatorial Test Execution

After the generation of a test suite, the tester has to stimulate the SUT with the given test inputs. As tests have to be executed frequently, modern software development processes call for automated test execution.

In the context of test automation, a *test script* encodes the sequences of steps used to stimulate the SUT and to observe its response [9]. Oftentimes, the same sequence of steps is executed again and again with varying test inputs. Then, data-driven testing is an appropriate abstraction to improve maintainability of test scripts. A *parameterized test script* describes a sequence of steps to exercise the SUT with placeholders (variables) to represent variation points [9]. They are then instantiated with varying test inputs. Thereby, the number of test scripts which must be maintained is drastically reduced.

An important part of testing is the *test oracle*, which provides the expected results for given test inputs. Based on the test oracle, a *test comparator* compares the expected and actual results to decide whether the behavior of the SUT conforms to the specified one for given test inputs [10]. In other words, the test comparator labels a test input as either passed or failed. We assume the presence of an automated test oracle and test comparator, even though that is an active research area on its own [10].

Frameworks [11] often play an important part in test automation. In general, frameworks mostly act as orchestrators of configurable *extension points*, for which a user can either choose between a selection of framework-provided options or provide custom developed options. As the orchestration of test suites is similar from test project to test project, it makes sense to extract it into a framework. The testers can then register their test scripts via defined extension points, such as the `@Test` annotation in JUnit5.

D. Combinatorial Fault Characterization

After testing the SUT with all test inputs, each test input is labeled as either passed or failed. But oftentimes, not all values of a failed test input are responsible for triggering the failure. Instead, a sub-combination of the values can be identified such that each test input that contains this sub-combination triggers the failure. In general, each value or combination of values that triggers a failure is denoted as a failure-inducing combination (FIC).

When a developer needs to repair a system that failed, FICs are of great interest because they can simplify the process of finding the responsible defect.

Therefore, combinatorial fault characterization is an additional activity that attempts to find FICs in the input space. Usually, just finding FICs is not enough. Instead, *minimal* FICs should be identified to better explain the cause of a failure. A FIC is minimal if removing one parameter-value assignment leads to the combination no longer being failure-inducing, i.e. there exists a possible test input which contains the reduced combination but passes the test [12].

There are multiple algorithms to identify minimal FICs, which we denote as *fault characterization algorithms* (FCAs). They can be divided into non-adaptive and adaptive ones.

1) Non-Adaptive Fault Characterization: Non-adaptive algorithms only consider the execution results of the initial test suite when identifying possible FICs. Since the information given by most test suites is not enough to discern minimal FICs, most approaches in this category adjust the combination strategy such that the initial test suite follows a certain distribution [13]. Dedicated combination strategies like *locating and detecting arrays* are used [14]. Due to the way test suites are constructed, non-adaptive fault characterization leads to larger initial test suites. Consequently, they introduce an execution overhead even if no test input leads to failure. Additionally, they are often constrained in either the number of failure-inducing combinations they can find, or in the size of possible failure-inducing combinations [15].

2) Adaptive Fault Characterization: In contrast, adaptive FCAs do not require dedicated combination strategies. They can be used in conjunction with *conventional* combination strategies that try to minimize the size of a test suite.

Similar to non-adaptive algorithms, the results of executed test inputs are used to isolate a minimal FIC. When the information of a test suite is not sufficient, additional test inputs are generated and executed to obtain more information [16]. The activities of isolation and generation are repeated until sufficient information is available.

As a result, the number of executions remains as low as possible if no test input triggers a failure. However, the repeating activities make the overall CT process more complicated.

III. Related Work

Since manually applying a combination strategy is virtually impossible, many tools exist to automatically generate test suites. Well-known examples are PICT [17], ACTS [18], and AETG [19]. Each of them takes an IPM in a custom textual format as an input and returns test inputs in a textual representation. In addition, each tool offers a number of advanced features like constraints, variable testing strength, or seeding.

CITLAB is another test generation tool [20]. It defines a common input and output format and extension points to integrate additional combination strategies. Its objective is to provide a platform that enables the comparison of different combination strategies.

These tools only automate the generation of test inputs. Other activities are not considered. To automate the test execution, a test automation framework like JUnit [5] can be used, and parameterized test scripts can be instantiated with the generated test inputs.

In contrast to dedicated generation and execution tools, there are already frameworks that provide an integration of the two activities. For example, JCUnit [21], NUnit [22], and Pairwise [23] can generate test inputs and directly execute them without importing and conversion. They are realized as extensions to test automation frameworks. The extension generates test inputs and repeatedly executes a parameterized test script written in the test automation framework. However, they are often tied to a particular combination strategy and often offer fewer options than stand-alone generation tools. For example, NUnit and Pairwise only support 2-wise combinatorial testing and JCUnit does not support seeding.

On the other hand, the integration of generation and execution has many advantages. When using separated tools, the generated test inputs must be imported from its textual representation and converted into appropriate objects to execute the parameterized test script. For instance, JCUnit is completely integrated with JUnit4 and no additional work is required. The frameworks allow the tester to define values of parameters as native programming language constructs like objects, which often makes custom conversion logic obsolete [21].

Another type of combinatorial testing tools are fault characterization tools. To the best of our knowledge only one such tool is publicly available: BEN [24]. BEN is a dedicated tool with its own input and output formats. For a given IPM and test execution results new test inputs are generated, which the tester then needs to execute again. The execution results of the new test inputs are again passed to BEN after execution. The process of generating new test inputs, executing them and supplying the results to BEN must be repeated until FICs can be determined.

While BEN as a dedicated tool allows to automate the generation of additional test inputs and the location of FICs, the integration with other tools adds complexity to the CT process. The IPM and execution results must be converted to BEN-specific formats and, more importantly, the additional test inputs must be executed. Since the process of executing test inputs and supplying test results can be repeated an indefinite number of times, it most likely involves manual activities. Even though each dedicated tool for generation, execution, and fault characterization automates an activity, there exists no holistic integrative solution which combines all activities in a fully-automated manner.

Category	Papers	Initial Input	Info Per Test Input
non-adaptive	[14], [13], [25], [26] [27], [15], [28], [29]	IPM	test result
adaptive	[16], [3], [30], [31], [32], [33], [12], [34],	IPM	test result
	[35]	IPM	test result + error info
adaptive*	[36], [37], [38]	IPM	test result

Table II
CLASSIFICATION OF PUBLISHED FCAS

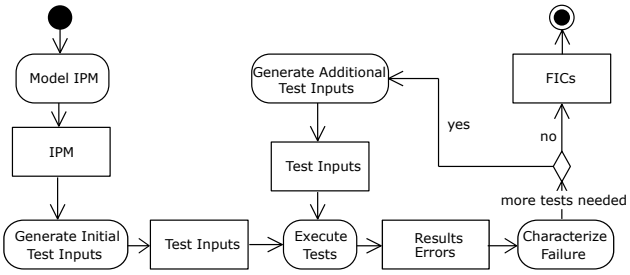

Figure 2. Complete CT & FC Process

IV. FRAMEWORK CONCEPT AND DESIGN

In the following, we present the core concepts and the top-level architecture of COFFEe, the **CO**mbinatorial test and **F**ault characterization **F**ram**E**work. In order to design a framework that supports all defined activities and is flexible enough to integrate existing as well as future CT and FC algorithms, we conducted a literature search to identify existing FCAs. Furthermore, we collected additional requirements the framework needs to realize. Based on the obtained results, we designed COFFEe's component-based architecture as follows.

A. Analysis of Fault Characterization Algorithms

Since many different FCAs exist, an important part of a general concept for an automated fault characterization framework is to assess the control flow and information needs of these algorithms. Therefore, we conducted a literature search to identify existing FCAs and to analyze them according to their adaptiveness and information needs.

Our literature search delivered 20 papers presenting FCAs (see Table II). We classified them and identified the information that is needed initially and per test input.

All FCAs only require the IPM (including testing strength) as input during initialization. In addition, the result of each executed test input, i.e. failed or passed, is required by all algorithms. One algorithm [35] also requires further error information like stack-traces to distinguish between multiple types of failures in one test suite.

As there are no significant differences regarding the needed information between adaptive and non-adaptive FCAs, the non-adaptive FCAs can be viewed as a specialization of the adaptive ones. Instead of generating more test inputs to narrow down FICs after the first test execution, they immediately calculate all FICs based on the initial test inputs. The difference is the generation of the initial test suite with a special combination strategy, e.g. via locating and detecting arrays [14]. Consequently, not only FC, but also CT algorithms must be exchangeable.

Analyzing the adaptive FCAs we noticed an import subcategory (denoted as adaptive*). These algorithms present an *interleaving approach* between initial CT and FCA. They only execute one test input at a time and, in case it fails, immediately enter a fault characterization loop to find the FIC responsible for the failure. As an advantage, these approaches require fewer test inputs, since any passed test input executed during FC contributes to the t-wise coverage. On the other hand, the classical approaches execute the complete suite of test inputs for this criterion regardless of any failing test inputs.

In the following, we concentrate on FCAs using the classical approach due to a stronger separation of concerns, and since more FCAs follow this approach.

Figure 2 depicts the general CT and FC process derived from the conducted literature search. The process starts with an initial test input generation, which provides test inputs to the test execution step. Next, an FCA decides whether to directly terminate the process and return all determined FICs, or to generate additional test inputs, which are executed to narrow down possible FICs.

B. Requirements

The overall goal of COFFEe is to coordinate and execute all automatable activities of the CT and FC process in order that a tester only needs to focus on providing the IPM and a parameterized test script. The framework takes over control and is responsible for test input generation, test execution, and fault characterization. To this end, the framework has to fulfill the following mayor requirements:

- **R1** The framework shall act as a mediator between the chosen CT and FC algorithms. Therefore, the framework should automatically switch between the activities depicted in Figure 2 when necessary, and should provide the required information to the used algorithms.
- **R2** Since testers want to perform different kinds of CT, the framework shall support the exchange of applied CT and FC algorithms.
- **R3** For better traceability and analyzability of the results, the framework shall offer reporting support of general process events, e.g. start and end of fault characterization, and of algorithm-specific events, e.g. FIC-likelihoods in BEN.
- **R4** As researchers or testers should be able to easily integrate new CT and FC algorithms, the framework shall offer an easy to apply extension mechanism (e.g. by means of simple interfaces).

C. Framework Architecture

Based on the requirements and the characteristics of the published FC algorithms, we designed the top-level architecture of the framework as depicted in Figure 3. All black-

coloured boxes indicate predefined framework extension points to integrate specific third-party components.

To decouple the features of the framework offered to manage IPMs and test scripts from those responsible for coordinating the applied CT and FC algorithms, we strictly separated both concerns by designing a *model* and an *engine* subsystem. The engine subsystem includes a component, *EngineLevelReporting*, for basic reporting and one, *GenerationManager*, implementing the logic to coordinate the algorithms. The model subsystem consists of the *TestModel* component, the *ModelLevelReporting* component and a component to convert between externally used formats to store e.g. IPMs and the engine-specific internal representation format (*Conversion*).

This design enables the creation of different test models using custom DSLs as well as new engines, e.g. for the interleaving FC approach.

D. Components of the Framework

The **TestModel** component is the primary entry point of the framework. It defines the format for the tester-supplied IPM and the format in which generated test inputs are passed back to the test script. In the beginning, the TestModel passes the given IPM to the *GenerationManager* after converting the IPM in its internal representation.

As different external representation formats should be supported, the **Conversion** component offers services to convert these into the internal format and vice versa.

The **GenerationManager** component is the most important one. It starts the initial test input generation using the chosen *GenerationAlgorithm*. In addition, this component decides if the fault characterization is performed after all respective test execution results are delivered by the *Executor*. In this case, the chosen FCA is triggered to generate new test inputs, which are propagated back to the *Executor*. Additionally, the *GenerationManager* provides useful process information to the *EngineLevelReporting* component in the internal representation format.

The **EngineLevelReporting** component consumes the information provided by the *GenerationManager* and additional algorithm-specific reports supplied through an interface available to all CT and FC algorithms. It then offers a single reporting interface to notify about all important events in the entire framework.

Since some reports, e.g. the propagation of all FICs, are done in the internal representation format, the framework contains a second component responsible for external reporting — the **ModelLevelReporting** component. It acts as an adapter between the internal reporting mechanism by registering itself with the *EngineLevelReporting* component, and then converts all reports into an external format.

E. Extension Points

The framework defines five extension points to adopt it and to integrate external components. One of these extension points offer functionality to outside of the framework, while the rest uses functionality provided by external components.

The **Orchestrator** extension point provides the framework's usage API. While the framework implements all functions to execute one given combinatorial test, it does not know where to find its configuration information, or how to deal with multiple tests. Therefore, an external orchestrator is responsible for determining all combinatorial tests which the framework should execute one by one, and how each of these tests is configured.

During a combinatorial test many test inputs need to be executed. Since the execution is test case specific, the tester needs to provide a parameterized test script using the **Executor** extension point. The framework then calls the given test script with the test inputs generated by the chosen CT and FC algorithms.

The initial generation of test inputs is done by an external **GenerationAlgorithm** component, that needs to provide an extension point to support arbitrary combination strategies. To adhere to approved design principles, the *GenerationAlgorithm* extension point is defined in the *GenerationManager* component, in order to avoid a dependency from inside the framework to an external component [11]. Since all combination strategies implement the same interface, they are interchangeable. Hence, testers can configure different combination strategies and can also provide their own strategies.

To use and configure FCAs, the **CharacterizationAlgorithm** extension point provides an interface to pass all information needed for fault characterization to a concrete FCA. This algorithm can then either return an identified FIC or a set of needed new test inputs.

If a tester wants to use a custom reporter, it can be implemented based on the interface provided by the **Reporter** extension point. The framework calls all registered engine-level reporters through an interface, and the *ModelLevelReporting* component converts all collected report items and propagates them to the registered model-level reporters.

V. COFFEE4J - A REFERENCE IMPLEMENTATION OF COFFEE

In the following, we present our Java reference implementation of the COFFEe framework, called *coffee4j*[1], which implements its architecture and smoothly integrates JUnit5. Conforming to the architecture, coffee4j consists of the subsystems `coffee4j-engine` and `coffee4j-model`. Additionally, the subsystem `coffee4j-junit-jupiter` consumes the services offered by both modules as a JUnit5 front-end.

Next, we introduce JUnit5, and subsequently take a more detailed look at some aspects of coffee4j's implementation.

A. JUnit5

JUnit5, a widely applied framework for test automation in Java, replaces its prior versions to enable features introduced in Java 8 and later [39]. JUnit5 differentiates between the general testing platform (*junit-platform*) and the concrete test engine (*junit-jupiter*), which is responsible for discovering

[1]coffee4j is publicly available via https://coffee4j.github.io.

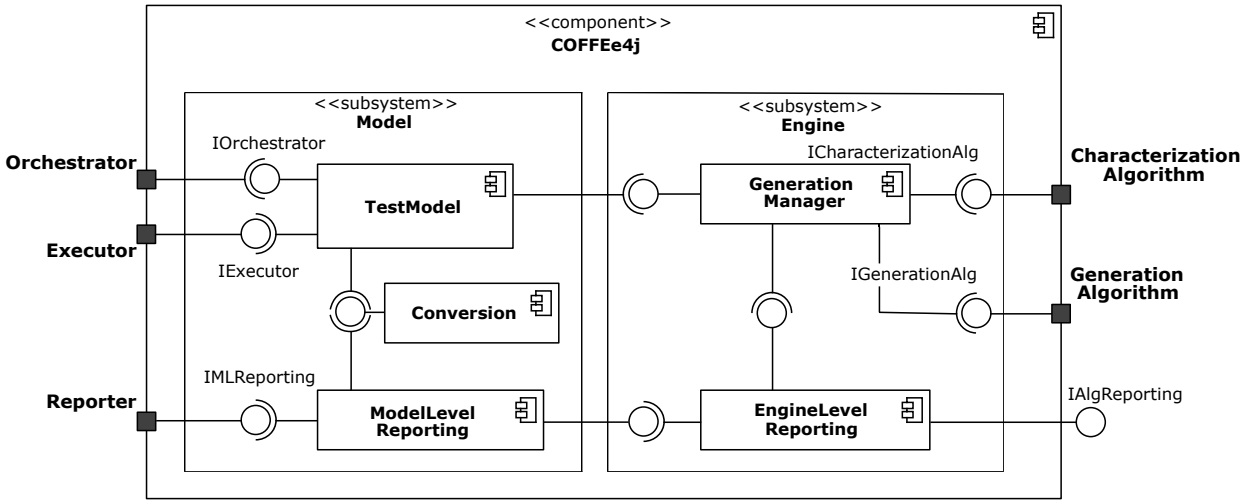

Figure 3. UML component diagram of COFFEe

and executing the test cases. The concept of parameterized test scripts is realized via Java methods. These so-called *test methods* are annotated with `@Test` so that they can be discovered and executed by the test engine. An extension for parameterized test methods is also provided by JUnit5. Then, `@ParameterizedTest` is used to annotate the test method which is parameterized via its method arguments [39]. In addition, a *source* must be configured, which provides a set of test inputs, and the parameterized test method is invoked for each test input.

For customization, JUnit5 offers an extension mechanism to register custom functionality for test classes and test methods, which JUnit5 calls at defined points in its life-cycle. Technically, JUnit5's extension mechanism consists of a number of interfaces whose implementations can be registered to test methods and test classes using defined Java annotations. Once JUnit5 reaches an extension point during execution, it calls the given methods of all registered extensions of the respective extension point with the current execution context. Using the execution contexts, a rudimentary communication between extensions can also be implemented.

All in all, JUnit5 provides two types of extension points. On one hand, it provides life-cycle callbacks at specific points of the test execution. On the other hand, it also provides more complex extension points, which are used to influence the actual execution of test methods. For example, a `TestTemplateInvocationContextProvider` extension can return a sequence of arbitrary test inputs for parameterized test methods in the form of so-called invocation contexts. A `ParameterResolver` can then resolve actual method parameters based on information contained in the contexts. Please refer to the JUnit5 documentation [39] for more information.

B. Implemented Extension Points

coffee4j realizes all defined framework extension points. In the following, we present the most important interfaces it provides, namely the ones for fault characterization and initial test input generation.

The following Listing 1 depicts the `FaultCharacterizationAlgorithm` interface, which realizes the *CharacterizationAlgorithm* extension point of COFFEe.

Listing 1. FCA Interface

```
interface FaultCharacterizationAlgorithm {
  List<int[]> computeNextTestInputs(
    Map<int[], TestResult> testResults);

  List<int[]>
    computeFailureInducingCombinations();
}
```

After executing the initial set of generated test inputs, the framework passes the obtained results to the `computeNextTestInputs` method. An adaptive FCA component, implementing this interface, can then decide which additional test inputs are needed and return them to the framework, while a non-adaptive one just returns an empty list. In the former case, the framework repeats the cycle of executing the current set of test inputs and providing the execution results to the FCA component until it returns an empty list. Next, the framework calls the `computeFailureInducingCombinations` method of the FCA component, that returns the computed FICs.

FCA components are initialized through a FCA factory, which provides the algorithm with an IPM. This enables to use multiple FCA components for different groups of test inputs, e.g. for positive and negative testing.

Listing 2. CT Interface

```
interface TestInputGroupGenerator {
  Collection<Supplier<TestInputGroup>>
    generate(CombinatorialTestModel model,
      Reporter reporter);
}
```

The `TestInputGroupGenerator` interface as shown in Listing 2 implements the *GenerationAlgorithm* extension point. Based on a `CombinatorialTestModel` and a reporter, each generator implementing this interface can return a number of groups, each containing multiple test inputs. This allows for a stronger separation of test inputs for different purposes on the framework level, e.g. negative and positive testing.

Instead of returning a collection of test inputs directly, `Supplier<TestInputGroup>` is an additional indirection in the `TestInputGroupGenerator` interface. The generation is encapsulated in a supplier, which is computed lazily on demand. Thereby, the generation of test inputs for different test input groups can easily be parallelized.

C. JUnit5 Extension

To offer the full benefits of JUnit5, e.g. a seamless integration in IDEs and build processes, coffee4j contains a collection of JUnit5 extensions to offer compatibility with the junit-jupiter test engine.

The annotation `@CombinatorialTest` is introduced to be used to annotate parameterized test methods for combinatorial testing. We denote them as *combinatorial test methods*. Each such method can have multiple annotations to configure how coffee4j generates the test input. This includes the used CT and FC algorithms. Additionally, the tester has to specify an IPM via the `@ModelFromMethod` annotation. It tells coffee4j which method to call to get the IPM used for the specific combinatorial test method.

Coffee4j's JUnit5 extensions then ensure the complete execution of the CT and FC process by calling the combinatorial test method with all test inputs, and using the results to generate further tests for FC. When running JUnit5, each combinatorial test method automatically registers the `CombinatorialTestExtension`, which has two responsibilities. Firstly, it loads the complete test configuration including the IPM via custom Java annotations (see example). The configuration is then used to start coffee4j's initial test input generation activity. Secondly, it converts the generated test inputs into a sequence of invocation contexts for the combinatorial test method. Therefor, it creates a Java stream, i.e. a lazily evaluated sequence, of test inputs using a `TestInputIterator`, which functions as the base source of test inputs. During the actual execution of the combinatorial test method by JUnit5, each test input of this stream is converted into an invocation context and finally the values of the test input are passed as arguments to the combinatorial test method.

For CT without FC, the concept of `TestInputIterator` would not be necessary. Every Java collection holding the initial test inputs can be converted to a stream. However, concurrent modification of the stream's source collection is not allowed during the execution, and therefore no FC test inputs could be added after the execution of the first test input began. The special iterator solves this problem by maintaining an internal queue of test inputs which need to be executed, and it only allows very basic operations on this queue hidden behind clearly defined public methods.

Once JUnit5 executes an invocation context, the context registers two additional context specific extensions: a parameter resolver, which passes the values of the context's test input to the arguments of the combinatorial test method, and a callback to propagate the execution result to coffee4j. Then, the framework can compute additional test inputs based on the results using the FCA component, and append them to the stream created in the combinatorial test extension for later execution.

For a minimal coffee4j combinatorial test setup with JUnit5, the tester has to (a) use the `CombinatorialTest` annotation on the defined test method and (b) define an `InputParameterModel` in a method which is linked to the test method via `@ModelFromMethod(<methodName>)`.

Listing 3 shows this configuration, also specifying a FCA and a reporter, which simply outputs all information to the console. It models the parameters and values from Table I. The combinatorial test method is left empty but normally contains instructions to exercise the SUT and check its output.

Of course, coffee4j's JUnit5 extension can handle more complex cases. If the defined annotations are not expressive enough to model a test configuration, a special annotation can be used to provide the complete test configuration via one custom method, thus giving complete freedom.

As the coffee4j-junit-jupiter implementation resolves parameters using the same external interfaces as junit-jupiter-params, testers can also reuse junit-jupiter-params specific features such as an `ArgumentAggregator` or `ArgumentConverter` [39].

VI. Application Experience

To demonstrate the flexibility and extensibility of coffee4j, we implemented the BEN [24], IterAIFL [30], AIFL [3] and Improved Delta Debugging [31] FCAs. Furthermore, we implemented different combinatorial test input generation algorithms; an IPOG-C strategy based on optimization and implementation advice from Kleine and Simon [40], and an extended algorithm to generate negative test inputs [41].

To evaluate the applicability of the overall framework, we applied the framework to the following three example systems with injected faults[2]:

- DRUPAL (47 boolean parameters and 45 constraints)
- BUSYBOX (68 boolean parameter and 16 constraints)
- LINUX (104 boolean parameters and 83 constraints)

The faults were injected via a special test comparator, which always lets the test fail if the input parameters contain specific combinations.

Tables III and IV show the results of applying the BEN and Improved Delta Debugging FCAs with some injected faults on the chosen systems. They name the used test *model* with the initial number of generated test inputs and needed generation time, the size of the identified *FIC*s (here x^y = y FICs of size x), needed execution *time* for FC, number of *found* FICs, number of *valid* reported FICs, and number of test *inputs*

[2]The models and injected faults are provided by Wu et al. [42].

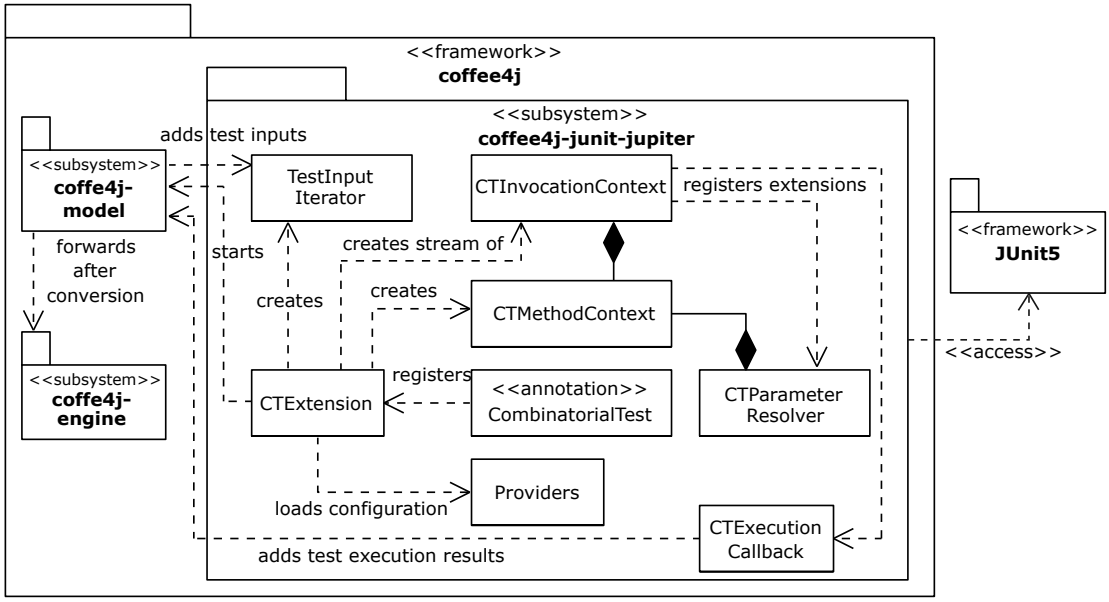

Figure 4. Architecture of coffee4j's JUnit5 extension

Listing 3. Coffee4jExample.java

```
class Coffee4jExample {
   @CombinatorialTest
   @CharacterizationAlgorithm(Ben.class)
   @Reporter(PrintStreamExecutionReporter.class)
   @ModelFromMethod("model")
   void combinatorialTest(String os, String browser, int ping, int speed) {
      // stimulate the SUT and check its output
   }

   private static InputParameterModel.Builder model() {
      return inputParameterModel("exampleTest")
            .strength(2)
            .parameters(
                  parameter("OS").values("Windows", "Linux", "MacOS", "Android", "iOS"),
                  parameter("Browser").values("Chrome", "Edge", "Firefox", "Safari"),
                  parameter("Ping").values(10, 100, 1000),
                  parameter("Speed").values(1, 10, 100, 1000));
   }
}
```

Model	FICs	Time	Found	Valid	Inputs
Busybox	1^2	12s 47ms	1900	1	20
18	2	453ms	1	1	61
10s 517ms	1	9s 562ms	1817	1	20
Drupal	2	219ms	1	1	39
17	2^5	1s 628ms	5	5	147
5s 164ms	2^5	2s 610ms	970	5	13
Linux	2	499ms	1	1	31
28	1	12s 440ms	1884	0	16
107s 633ms					

Table III
BEN EXECUTION RESULT

Model	FICs	Time	Found	Valid	Inputs
Busybox	1^2	58ms	2	2	12
18	2	93	1	1	10
10s 517ms	1	76ms	1	1	6
Drupal	2	77ms	1	1	9
17	2^5	79ms	2	1	17
5s 164ms	2^5	78ms	2	1	12
Linux	2	94ms	1	1	12
28	1	64ms	1	1	7
107s 633ms					

Table IV
IMPROVED DELTA DEBUGGING EXECUTION RESULT

needed for FC. Coffee4j was executed within the JetBrains IntelliJ IDE on a AMD FX-6100 CPU with 16 GB of DDR3 RAM, using Java 8.

The number of FICs identified by BEN is so high because of the specific implementation and the way the algorithm is designed. BEN maintains an internal list of all possible FICs of size t. An internal function assigns a probability of actually being failure-inducing to each possible FIC. However, when the actual FIC consists of fewer values than the testing strength t, all t-sized combinations which that contain the FIC are considered as failure-inducing. Consequently, BEN keeps them in its internal list. Since our implementation simply returns all possible FICs of size t and calculates which smaller ones could also be failure-inducing, many combinations are returned in such a case. By introducing a limit to the number or probability of returned FICs the number would be much lower while still maintaining a high percentage of correctly discovered FICs.

As can be noticed, both algorithms found the correct FICs in many cases, with Improved Delta Debugging requiring less time and fewer test method executions. However, this result cannot be generalized, since the injected faults were not chosen for a systematic comparison of FC algorithms.

We also tried to execute the scenarios with AIFL and IterAIFL, but both algorithms failed with an `OutOfMemoryError`, because they maintain an internal list of *all* possible FICs, not only of those with a size smaller than the testing strength. Since the models contain up to 104 parameters, execution was not possible, as this would require an internal list containing up to 2^{104} items, which results in an Java exception as the heap is too small.

All in all, the application of coffee4j is promising and clearly demonstrates that coffee4j can effectively integrate different CT and FC algorithms as well as test models. It ensures the correct execution of the given FCAs and initial test input generation algorithms.

VII. Conclusion

Test automation is an important part of current software development. While some tools offer the isolated ability to automate combinatorial test generation, test execution or fault characterization, yet no tool exists that integrates all three activities in one automated process.

When using tools with isolated abilities, the tester has to copy and transform information between test generation, test execution, and fault characterization tools. The process becomes tedious and error-prone. Therefore, it would be desirable to have an integration of all activities, namely input parameter modelling, test generation, test execution, and fault characterization. Then, the tester only models the input parameters and develops executable test scripts. A tool that integrates all activities could automatically handle the generation of test inputs, execution of the test scripts , and fault characterization, i.e. the computation of FICs and, if needed, the generation and execution of additional test inputs.

The objective of this paper was to create a framework that integrates all these activities in a extensible fashion such that different algorithms for CT and FC can be developed and used.

Therefore, we first summarized the current state of research for tools, and clearly identified the lack of an automated tool covering all three activities of the CT process and motivated the necessity of integrating all three activities. In a first step to close this gap in available tools, a literature search examined current FCAs according to their needed information and categorization into non-adaptive and adaptive algorithms. Next, we presented the concept of a general architecture named COFFEe - a combinatorial test and fault characterization framework. Based upon collected requirements, COFFEe supports a strong separation of test model and test engine related functions by defining respective subsystems.

Furthermore, we presented coffee4j, which is a Java implementation of the proposed COFFEe framework. In coffee4j, we also provided a JUnit5 integration, allowing for a seamless integration of combinatorial test generation and fault characterization into IDEs and automated build processes.

Finally, we demonstrated the capabilities of COFFEe and coffee4j by applying it to different test scenarios using different algorithms and different injected faults. It showed that coffee4j is able to effectively integrate different algorithms for CT and FC However, the different algorithms and injected faults showed differences in execution time and found FICs. To determine which FC algorithms performs best under which conditions, further investigations are required.

As future work, we will extend the framework to support interleaving approaches as well, and we will also implement the other identified FCAs. Then, we plan to design experiments and to utilize the framework in order to conduct a comparison among the different FCAs.

References

[1] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, June 2004.

[2] K. Fögen and H. Lichter, "A case study on robustness fault characteristics for combinatorial testing - results and challenges," in *2018 6th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2018) co-located with APSEC 2018*, 2018.

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

[4] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[5] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Phillip, and C. Stein, "Junit5," https://junit.org/junit5/, 2015.

[6] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press, 2007, pp. 255–260.

[7] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

[8] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: A general strategy for t-way software testing," in *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*. IEEE, 2007, pp. 549–556.

[9] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[11] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[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] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Algorithms to locate errors using covering arrays," in *LATIN 2008: Theoretical Informatics*, E. S. Laber, C. Bornstein, L. T. Nogueira, and L. Faria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 504–519.

[14] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," *Journal of Combinatorial Optimization*, vol. 15, no. 1, pp. 17–48, Jan 2008.

[15] 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)*, July 2017, pp. 138–149.

[16] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn, "Fault localization based on failure-inducing combinations," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2013, pp. 168–177.

[17] J. Czerwonka, "Pairwise testing in real world," in *24th Pacific Northwest Software Quality Conference*, vol. 200. Citeseer, 2006.

[18] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 370–375.

[19] 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*, vol. 23, no. 7, pp. 437–444, July 1997.

[20] P. Vavassori and A. Gargantini, "Citlab: A laboratory for combinatorial interaction testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation(ICST)*, vol. 00, 04 2012, pp. 559–568.

[21] H. Ukai and X. Qu, "Test design as code: Jcunit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 508–515.

[22] C. Poole and R. Prouse, "NUnit," http://nunit.org/docs/2.6.4/pairwise.html, 2009.

[23] RetailMeNot, Inc., "Pairwise," https://github.com/RetailMeNot/pairwise, 2014.

[24] 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*. IEEE, 2015, pp. 1–4.

[25] 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.

[26] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 177–188.

[27] W. Zheng, X. Wu, D. Hu, and Q. Zhu, "Locating minimal fault interaction in combinatorial testing," *Advances in Software Engineering*, vol. 2016, 2016.

[28] J. Zhang, F. Ma, and Z. Zhang, "Faulty interaction identification via constraint solving and optimization," in *Theory and Applications of Satisfiability Testing – SAT 2012*, A. Cimatti and R. Sebastiani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 186–199.

[29] 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 (ICSTW)*, March 2017, pp. 276–277.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

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

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

[38] 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.

[39] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Phillip, and C. Stein, "Junit 5 user guide," https://junit.org/junit5/docs/current/user-guide/, 2018.

[40] 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, Mar. 2018.

[41] K. Fögen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *2018 IEEE Eleventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 7th International Workshop on Combinatorial Testing (IWCT)*, 2018.

[42] 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.