Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Feature-Trace: An Approach to Generate Operational Profile and to Support Regression Testing from BDD Features

Rafael Fazzolino P. Barbosa

Dissertação apresentada como requisito parcial para conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2020

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Feature-Trace: An Approach to Generate Operational Profile and to Support Regression Testing from BDD Features

Rafael Fazzolino P. Barbosa

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 27 de January de 2020

Acknowledgements

First of all, I thank my mother, Monica Fazzolino Pinto, for all the support and understanding during my walk here. To my family as a whole, especially my girlfriend Letícia, who endured and supported me in all difficult times during this phase of my life, and to all my friends who also supported me. I also thank the University of Brasilia as a whole, which today I see as a second mother, despite many difficult moments, I can only be grateful.

I want to highlight a special thanks to my advisor, Genaína Nunes Rodrigues, who introduced me to the world of academia, guiding my steps during each activity over the years that we have worked together, both in research projects and in master's activities. I am very grateful for the patience and understanding during difficult times that we have spent in these years. I am pleased to have worked with such a good teacher, both as a researcher and as a person.

Abstract

Operational Profiles provide quantitative information about how the software will be used, which supports highlighting those software components more sensitive to reliability based on their profile usage. However, the generation of Operational Profiles usually requires a considerable team effort to liaise requirements specification until their reification into expected software artifacts. In this sense, it becomes paramount in the software life cycle the ability to seamlessly or efficiently perform traceability from requirement to code, embracing the testing process as a means to ensure that the requirements are satisfiably covered and addressed. In this work, we propose the Feature-Trace approach which merges the advantages of the Operational Profile and the benefits of the requirements-to-code traceability present in the BDD (Behavior-Driven Development) approach. The primary goal of our work is to use the BDD approach as an information source for the semi-automated generation of the Operational Profile, but it also aims to extract several other metrics related to the process of prioritizing and selecting test cases, such as the Program Spectrum and Code Complexity metrics. The proposed approach was evaluated on the Diaspora software, on a GitHub open source software, which contains 68 BDD features, specified in 267 scenarios and $\approx$ 72 KLOC and more than 2,900 forks and counting. The case study revealed that the Feature-Trace approach is capable of extracting the operational profile seamlessly from the specified Diaspora's BDD features as well as obtaining and presenting vital information to guide the process of test cases prioritization. The approach was also assessed based on feedback from 18 developers who had access to the approach and tool proposed in this work — making evident the usefulness of the Feature-Trace for activities of "Prioritization and Selection of Test Cases", "Evaluation of the quality of test cases" and "Maintenance and Software Evolution".

Keywords: Operational Profile, Behavior Driven Development (BDD), Regression Tests

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is well known that the software development process is not limited to development activities, just generating a product and delivering it to the customer. The software product should be considered as something that continually evolves and adapts from the requirements elicitation step, starting the software product, to the maintenance and evolution of the software product, which we call the software life cycle. [3].

Throughout the life cycle, we find several challenges that can harm the result, generating a product with many failures or even failing to complete a deliverable artifact. Thus, techniques and approaches that facilitate the management and monitoring of the software development, maintenance, and testing process are being discussed and applied by the computing community around the world, as noticeable in [4, 5, 6].

1.1 Team Challenges Throughout the Software Lifecycle

The software lifecycle involves several challenges that need to be overcome by the team for the product to be delivered, maintained, and evolved successfully. The challenges start from the first stage of the process, in the requirements elicitation phase, to the last stage, in the software maintenance and evolution phase. The causes of the problems can vary widely. According to [7], most of such problems are the result of human errors during the development process. Thus, the software lifecycle is subject to the inclusion of failures at all times, even in experienced and skilled teams.

Because failure inclusion can occur at any time during the software lifecycle, the project team must carefully manage and monitor the evolution of the source code, seeking to know the software product as a whole. [8]. When we speak of “knowing” the software product, we refer to the relationships between each product level, from the requirements, the source code that implements these requirements, and the test cases that verify the operation of this source code based on the requirements. From these relationships, one can track certain software functionality at different levels: requirements, source code, and test cases. In addition, it is essential to know how the end-user will use the delivered software product, making it possible to distinguish, among the software entities, the degree of business importance to the customer [1].

Software product traceability might benefit the entire software lifecycle [9]. This is particularly important in the software validation and verification activities.

1.1.1 Software Verification and Validation Context

The activities of Verification and Validation identify failures so that they are corrected before the delivery of the system. According to [10], Verification and Validation activities must occur during the entire system development cycle, seeking to identify failures as soon as possible, saving time and money. These activities are usually automated from the implementation of test cases, making it possible to generate, for example, regression test suites that can check the software whenever a new change is made [8].

Test cases are usually classified in white- and black-box testing, where the first is more related to the activity of Software Verification, also known as structural testing. On the other hand, the black-box tests seek to validate the software product, that is, check if the software meets the requirements elicited [10].

Given the notorious challenge regarding the verification and validation of software completeness through test cases [11], it is paramount to delimit a relevant set of test cases that guarantees the highest possible reliability within the limitations of the software development team [8]. Aiming at this goal, several techniques for prioritizing test cases have been proposed, related to the prioritization of either white- or black-box testing or both [12]. The prioritization process of test cases can occur from several strategies. These strategies use the data from the system under test (SUT) to define the prioritization criteria of test cases, such as the complexity of code units (effort to test that unit) [13], Operational Profile (OP) information [1], history of failures [14] and similarity of the test cases [15], among others [11]. Overall, these strategies focus mostly on software verification and validation activities, which seek to ensure that the software is being developed in the right way and that the correct software is being developed, respectively [8].

In particular, many techniques are based on concepts such as OP, which takes into account the probability of execution of each entity of the software to provide quantitative information about how the software will be used [16]. There are several advantages of using the OP in the software development cycle: (1) to reduce costs with operations with little impact on system reliability; (2) to maximize the speed of development, from the proper allocation of resources, taking into account the usage and the criticality of the operations; (3) to guide effort distribution in requirements, code, and design revisions, and (4) to make testing more efficient (greater slope in system reliability growth rate). However, there are several limitations that often render impractical the use of the OP in the software development cycle. The effort and the complexity involved in generating the OP are considerably high as well as the tendency to the stability of the reliability growth during the process of prioritizing test cases. That is, after a certain number of test cycles, the growth rate of reliability tends to be stable due to the non-prioritization of units that have a low probability of occurrence. [17].

It is worth noting that the quality of software products involves not only the quality of the source code but also the quality of other software artifacts, e.g. requirements specification and tests suites, and the traceability between such artifacts [9]. Moreover, the success of software testing activity is directly connected to the correct alignment between requirements and test cases [18]. The lack of traceability between artifacts can lead a software project to complete failure [9]. Communication between requirements engineers and test engineers is often compromised by dealing with abstraction levels, languages and entirely unconnected complexities [9]. Thus, several studies have proposed strategies and techniques for locating requirements in the source code [19, 20, 21, 22, 23].

Some approaches have emerged to bring the requirements artifacts closer together and Verification and Validation. In this sense, it is worth highlighting the methodologies of agile development, which have sought to minimize the gap between the Requirements activities and the Verification and Validation software activities [24]. Some practices of the agile development process deserve to be highlighted when we refer to the gap between the activities of Requirements Elicitation and Verification and Validation, such as:

- *Face-to-face communication*
- *Customer involvement and interaction*
- *User stories*
- *Change management*
- *Review meetings and acceptance tests*

Among the approaches used to minimize this gap and which are gaining notoriety is the *Behavior Driven Development* (BDD) [25]. In BDD, system requirements and test cases merge into the same artifact, making the requirement automatically verifiable and enabling a familiar interface among all stakeholders in the software project. The approach uses a ubiquitous language for writing artifacts in the form of user stories. However, the adoption of the BDD approach also involves some challenges, such as (i) poorly written scenarios, i.e. scenarios lacking comprehensive requirements representation, which may render the test assurance process weak and (ii) difficulty in convincing the stakeholders to use BDD, since it requires a paradigm shift from the stakeholders to specify the features in BDD language [26]. Moreover, BDD artifacts do not provide quantitative information to support test case prioritization, unlike OP.

1.2 Research Problem

Taking into consideration the context presented above, it is observed the importance of methodologies, techniques, and approaches that facilitate the process of quality assurance of the software product developed throughout its life cycle. Several strategies have been analyzed and used by the computing community worldwide. The software community also often uses various metrics, tools, and approaches to improve the development process, with OP and BDD being essential examples in this context, and may also cite data regarding the complexity of the source code.

In this sense, there is a need to centralize some of these techniques in such a way as to enable a debate between those involved, with access to relevant information from each software product entity, as well as traceability between software artifacts. Thus, we highlight as the main problem attacked in this research: *The difficulty in obtaining and cross-referencing software artifact information to enable the distribution of product knowledge as a whole, from overall traceability to metrics for each product entity.* By enabling knowledge about software artifacts among all involved, we allow debates about the importance of each method, requirement, or test case. With this, the process of prioritization and selection of test cases becomes a joint activity in which all participants can analyze the context from various points of view, enriching the debate and thereby enhancing the process of Verification and Validation. Also enriching all other activities involved in the software life cycle.

As a result, we come to the following research question:

Can BDD artifacts be used to generate the OP and guide a prioritization and selection testing process?

There are several limitations that often render impractical the use of the OP in the software development cycle. The effort and the complexity involved in generating the OP are considerably high as well as the tendency to the stability of the reliability growth during the process of prioritizing test cases. That is, after a certain number of test cycles, the growth rate of reliability tends to be stable due to the non-prioritization of units that have a low probability of occurrence [17]. Moreover, the success of software testing activity is directly connected to the correct alignment between requirements and test cases [18]. The lack of traceability between artifacts can lead a software project to complete failure [9]. Communication between requirements engineers and test engineers is often compromised by dealing with abstraction levels, languages and entirely unconnected complexities [9].

1.3 Research Contribution

Given the gaps and difficulties presented to ensure the traceability between requirements and software testing, the lack of an approach that provides a solid way of applying the concepts inherent to the OP, and the difficulty in crossing some metrics that support the prioritization and selection of test cases, we propose the *Feature-Trace* approach.

To this end, we come to the following contributions of this work:

1. A literature review on quality assurance activities throughout the software life cycle, with a primary focus on prioritization and selection of test cases;
2. A test case prioritization and selection approach that utilizes a variety of information extracted from source code. With that, allow the prioritization and selection of test cases;
3. A framework to extract the necessary data and present the full set of information, enabling the use of the proposed approach in work;
4. A case study seeking to identify the behavior of teams using the information presented by the Feature-Trace framework during the process of prioritization and selection of test cases;
5. Analysis and reporting the outcome of the case study, sharing the knowledge generated with the worldwide software engineering community (this work and [27]).

Our Feature-Trace approach facilitates the understanding of the software product from the perspective of the tested requirements as BDD artifacts and their attributed relevance. Then, leveraged by the construction of the OP from the BDD artifacts, Feature-Trace enables the traceability, testability and quantitative measure of importance from the requirements to the code units and their corresponding test cases. By these means, from the BDD artifacts and the SUT source code, relevant information is seamlessly extracted to guide the software testing process.

The proposed approach was evaluated on the Diaspora social network software, an open source project, which contains 68 BDD features, specified in 267 scenarios. Diaspora has approximately 72 thousand lines, with 2,915 *forks* and 12,226 *stars* in its repository in Github[1]. The case study revealed that the Feature-Trace approach is capable of extracting the OP as well as obtaining and presenting vital information to guide the prioritization process of test cases. Also, a study with 18 participants was carried out to analyze the feedback of these participants on the usefulness of using the approach and Feature-Trace tool. It was observed, as presented in Chapter 4, that, according to the participants, the proposed approach has a high degree of utility in the questions "Prioritization/Selection of test cases", "support in activities of Software Maintenance and Evolution" and the "ability to represent the current situation of test cases".

In a nutshell, the contributions of this work are:

1. An approach that allows for: (1) the creation of OP seamlessly stemmed from well-formed, executable and traceable requirements and (2) the prioritization of software testing based on the created OP as well as on a set of quantitative SUT information, i.e, program spectrum, number of impacted software features and code complexity.
2. A tool that automates the proposed approach, allowing the application of the approach without great effort by the testing team primarily based on the BDD scenarios specifications, source code implementation and test suite.
3. A case study on the Diaspora social network as a GitHub open project that shows the applicability and the potential of our approach.
4. A case study with eighteen developers that used the tool and shared some of the perceptions gained by each participant.

1.4 Organization

The remaining sections are organized as follows: Chapter 2 presents key background concepts. Chapter 3 provides an in-depth perspective of our Feature-Trace approach and tool. Chapter 4 presents the diaspora case study and the perceptions of each developer who used our tool and approach to prioritize and select test cases. Chapter 5 compares our approach to the most related work. Finally Chapter 6 concludes the work with some discussions, lessons learned and outlines future works.

[1]https://github.com/diaspora/diaspora

Chapter 2

Background

In this chapter, the concepts necessary to understand this work are presented. Initially, the Verification and Validation context is discussed, as well as some coverage metrics. Then we present the concepts of Operational Profile and Behavior Driven Development, as well as a brief presentation of the Program Spectrum concept.

2.1 Software Verification and Validation

According to [28], "*Validation of software or, more generally, verification and validation (V&V), intends to show that software conforms to its specifications while satisfying specifications of the system client.*". In this way, the Software Verification and Validation process contribute directly to a successful project, which makes this activity so important in the software development process.

The concept of Verification and Validation is linked to the questions: (1) "*Am I building the system correctly?*" (verification) and (2) "*Am I building the correct system?*" (validation) [29]. One of the main reasons for unsuccessful software delivery involves these two issues since often the customer himself does not know exactly what he expects from the required software product. In addition to this problem, one must also consider the way the software is being implemented, taking into account the quality of the generated code, using metrics such as Cyclomatic Complexity, readability, maintainability, and testability, for example. Concerns such as these are part of the Software Verification and Validation process (V&V).

The primary strategy used to maximize the efficiency of this activity is the implementation of automated test cases, guaranteeing the suitability of the system to the requirements established with the client, as well as guaranteeing specific criteria of verification of generated code [29].

In order to measure the quality of the tests, which encompass a large part of the Verification and Validation process of the system, several techniques and metrics are used. The following is the Regression Test activity, highlighting the main approaches that seek to minimize the effort of this activity.

[29] states that the main strategy used to maximize the efficiency of this activity is the implementation of automated test cases, ensuring the adequacy of the system to the requirements established with the client, in addition to guaranteeing certain criteria for verifying the generated code. According to the author, tests can be classified into the following categories:

- *Unit Testing*:

  Responsible for ensuring the operation of each unit of the system.

- *Integration Test*:

  Responsible for ensuring the functioning of each module of the system, checking the interface between the units of the system.

- *Acceptance Test*:

  Responsible for ensuring the correct functioning of the requirements specified with the customer. It is considered a type of black box test.

- *System test*:

  Responsible for ensuring the operation of the system in its real context, that is, taking into account the *hardware* used to run the system. It is also considered a black box test.

Due to the great importance of the V&V activity, several strategies and techniques have been raised seeking to maximize the efficiency of this activity, as shown by [18], where it analyzes verification and validation strategies in six large companies. The main challenge analyzed in the study involves the concept of traceability between requirements and the system validation process, since, according to the author, the lack of traceability between requirements and V&V can generate serious problems related to deadlines and quality of deliverable.

The issue of traceability between requirements and test cases is attacked by the *Behavior-Driven Development* approach, as shown by [30]. This approach is used in this research in order to minimize the *gap* between the requirements of the system and its verification and validation process, minimizing several problems such as those previously presented.

Seeking to measure the quality of the tests, which covers a large part of the system's Verification and Validation process, several metrics are used to measure test coverage. Some of these metrics are described in the following section.

2.1.1 Coverage Metrics

The main purpose of testing coverage is to ensure that each requirement is being tested at least once, in this case making reference to black box tests. According to [31], a set of black-box tests can generate low coverage according to some criteria 1) If there are implicit or unspecified requirements; 2) In the existence of code not derived from the requirements or; 3) When the set of test cases derived from the requirements is inadequate. Problems like these can be solved with the traceability between requirements and test cases, as in the strategy used by the BDD [32] approach. The main coverage metrics surveyed by [17] are:

- *Functional Coverage*: According to [33], *Functional Coverage* can be defined as the criterion responsible for ensuring that all system functions (or methods) were exercised during the execution of the test case set. The coverage obtained refers to the percentage of functions/methods exercised in relation to the total number of functions/methods in the system.
- *Branch Coverage*: The coverage criterion of *branches* seeks to ensure that all code flow control points are exercised during the execution of the test cases [34]. According to [35], the *branch* coverage criterion is a more impacting criterion than the *Statement Coverage*, since if all *branches* of the system are exercised, necessarily all the system declarations will also be executed, that is, if a set of test cases meets the criteria of *Branch Coverage*, this same set it also meets the *Statement Coverage* criterion.
- *Statement Coverage*: According to [34], the statement coverage is the basic criterion for analyzing system test coverage. However, it is a metric quite used and accepted by the community in general, defined by the ANSI *American National Standards Institute*. This coverage criterion takes into account the execution of each statement throughout the system code, that is, verifying which statements were executed by the set of test cases, following a strategy very similar to the last two coverage criteria.
- *Operational Coverage*: The concept of *Operational Coverage* was proposed by [36], looking for a coverage criterion that fits as a stop rule in the test process based on the Operational Profile of the system. This strategy takes into account the fact

that certain failures have a greater impact on system reliability than other failures, according to the Operational Profile in question [16].

The definition of *Operational Coverage* takes into account the frequency of use of each entity based on the concept of *Program Spectrum* [37], distributing each entity in the system into groups of importance, to which they are assigned weights, as suggested by [36]. The assignments are made in such a way that, the greater the frequency of use of the entity, the greater the weight of the group to which it belongs, making clear the degree of impact on the system's reliability. The value of the *Operational Coverage* (OC) of the system is obtained from the Equation (2.1).

$$OC = \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

where,

n = number of importance groups used;

x_i = rate of entities covered in the group $_i$;

w_i = group weight $_i$.

2.1.2 Regression Testing

During the process of development, maintenance and evolution of software, a test suite needs to be maintained and evolved, allowing the execution of test cases whenever changes are made in the source code, guaranteeing the correctness of the features and avoiding the inclusion of defects in the software. This type of test is called the Regression Test and may involve some difficulties over time. According to [11], regression testing can be a costly and slow activity, taking days or even weeks to run, depending on the size of the test suite in question.

In order to minimize the effort involved in the regression test activity, [38] highlights three main approaches: (1) test suite minimization, (2) test case selection, and (3) test case prioritization.

- *Test Suite Minimization (TSM)* Seeks to remove redundant test cases to reduce the size of the test suite, and thereby minimize the time required for its execution. It also tends to reduce the ability of the test suite to identify defects in the software [38]. In this way, the reduction of the test suite should be performed with care to avoid the removal of important test cases.

- *Test Case Selection (TCS)* Selects a particular subset of test cases that is most likely to identify a new defect in the software. Generally, the set of test cases selected is composed of test cases that exercise the modified source units during the last modification of the software. It has a similar strategy to the test suite minimization, that executes only a subset of the test suite. However, the non-executed test cases are not removed from the test suite, waiting for the moment when they will be selected in a future execution.

- *Test Case Prioritization (TCP)* Seeks to order the execution of the test cases in such a way as to maximize certain properties, such as the failure identification rate. Thus, with the prioritization of test cases, it is expected that the most critical test cases are performed first.

The three approaches outlined above use a variety of techniques and strategies, based on source code metrics, test histories, software requirements, and more [11]. A well-known metric that can be used as a criterion for the three approaches is the Operational Profile concept, which will be presented in Section 2.2.

2.2 Operational Profile

The Operational Profile aims to quantify the probability of occurrence of each entity of the software, making it possible to obtain the importance of each entity when it refers to the impact on the reliability of the software [16].

Figure 2.1 presents the composition of the *Operational Profile* according to Musa [1].

The author defines a detailed process for creating the Operational Profile, containing the topics presented in Figure 2.1, which may include one or more of the following steps:

1. Define the *Customer Profile* It is the profile of an individual, group, or organization that is acquiring the software, being represented by an independent vector of types. Each *customer type* encompasses clients who similarly use the system. It is defined as a set of pairs "*customer type*/probability", making clear the relationship of importance between each type of customer, when referring to the reliability of the delivered system.

2. Define the *User Profile* It is the profile of a person, group, or institution that operates the system, and can be grouped into types of users. Likewise, a *User profile* is defined as a set of pairs "*user type*/probability".

3. Define the *System-mode Profile* Several software systems have distinct modes of operation, modifying the flow of execution as a whole. The *system-mode profile*

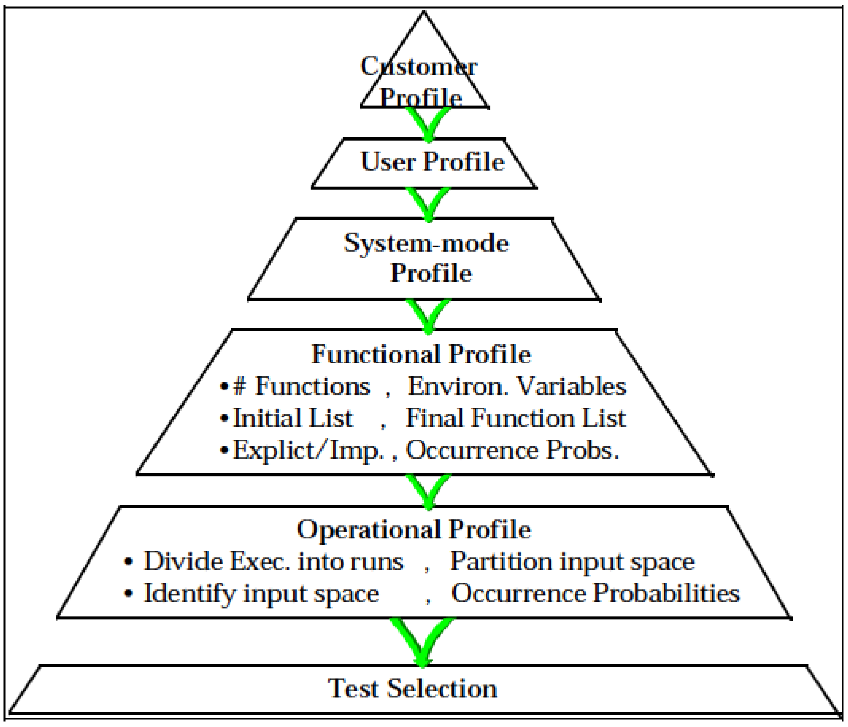

Customer Profile
User Profile
System-mode Profile
Functional Profile
•# Functions , Environ. Variables
•Initial List , Final Function List
•Explict/Imp. , Occurrence Probs.
Operational Profile
• Divide Exec. into runs , Partition input space
• Identify input space , Occurrence Probabilities
Test Selection

Figure 2.1: *Operational Profile* composition [1].

tries to map the degree of importance of each mode of operation of the system in relation to the reliability of the system. Software that can be executed in both graphics mode and batch mode, for example, has two *system-mode profiles*: 1) batch mode and 2) graphic mode.

4. Define the *Functional Profile* Represents the use profile of features that an external entity, such as a user, can perform on the system. The functionalities are established during the elicitation of the requirements, thus, obtaining the functionalities should be a simple activity of reading the system requirements, considering, of course, that the requirements are reliable and represent the system as a whole.

   The functional profile can be either **explicit** or **implicit**, depending on the representation of the possible inputs present in this functionality, called **key-input variable**. This distinction is required because of the behavior of the feature according to each input value range received. Depending on the functionality, the execution path of the system can be modified according to the input value, demonstrating a distinct behavior, that is, requiring an isolated treatment in obtaining the *functional profile*.

   The **explicit** profile requires the specification of each range (also called the level) of each variable of the functionality in question, requiring a significant effort and obtaining a higher number of elements for *Operational Profile* mapping. On the other hand, the *implicit* profile is specified in *subprofiles* of each key variable. That is, each range is related to its probability within its *subprofile.* In the *explicit* profile, the number of elements is equal to the product of the number of levels per variable, while in the *implicit* profile the number of elements is similar to the sum of the number of levels with the number of variables present.

To create the *Functional Profile*, [1] suggests the following order of activities:

(a) **Generate an initial list of functions**: This list can be easily obtained from the analysis of system requirements. If the requirements are superficial, this information can be obtained from the project stakeholders.

(b) **Raise Environment Variables**: It characterizes conditions that influence the execution path of the system within the same functionality.

(c) **Define final list of functions**: The final list (FL) of functions consists of the functionalities and their own environment variables, for each function that occurs in the system. The number of functionalities/variables is calculated from equation (2.2).

$$FL = (n \times l) - (\bar{n} \times \bar{l}) \tag{2.2}$$

where,

n = number of functions in the initial list;

l = total number of levels of all environment variables;

$\bar{n}$ and $\bar{l}$ = combination of functions and environment variables that do not occur, respectively.

(d) **Distribute probabilities**: The distribution of probabilities can be done in several ways. However, it is known that the ideal context is the distribution be made from the analysis of data of the use of the system in a production environment, or of a system similar to this one. However, access to this information is often not feasible for a variety of reasons. In situations where it is not possible to obtain historical data on the use of the system, it is possible to estimate the *functional profile*, from the interaction with users and experts in the business context that the system will be enveloped.

5. Define the *Operational Profile*

While the *Functional Profile* is a representation of the capabilities of the system from a user's point of view, the *Operational Profile* is a representation from the developers' point of view. In this way, the operations present a clearer vision for who will implement the system tests. According to [1], the list of operations can be extracted from the *Functional Profile*, defining a one-to-many relationship between features and operations.

According to [1], the number of operations resulting from the *Operational Profile* creation process can reach a very high value, rendering the test process unfeasible. Thus, the author suggests the reduction of some of the elements detailed above, according to the analysis of the expert involved in the process of creating the OP.

Among the main advantages of using the Operational Profile are:

- Reduce costs with operations with little impact on software reliability;
- Guide effort distribution into requirements, code and design reviews;
- Make testing more efficient (greater slope in the growth rate of system reliability);
- Make management more accurate;
- Guide the creation of user manual and training documentation.

2.3 Behavior Driven Development

The BDD approach is a derivation of the concepts of TDD (Test Driven Development) [39] and DDD (Domain Driven Design) [40], including an easy-to-understand language that enables all stakeholders to understand and collaborate with the writing of the requirements of the software.

Among the ubiquitous language formats used by the BDD approach, we highlight *Gherkin*[1], which has a set of syntactic rules that allow the writing of scenarios of use that can be mapped to source code methods and executed automatically to validate their behavior, with the support of the appropriate tool.

Each *feature* is detailed in a *.feature* file, and may have one or more use scenarios of the *feature*. Listing 2.1 illustrates the reserved words and meta-defined steps of a BDD scenario.

Listing 2.1: Specification of a BDD feature in the Gherkin language

```
Feature: feature description
```

[1]https://cucumber.io/docs/gherkin/

```
Background: Common steps
Scenario: Some usage scenario
  Given some precondition
  And some other preconditions
  When some action occurs
  And some other action occurs
  And yet another action
  Then some testable outcome is obtained
  And something else that we can check
```

The items below present the definition of the keywords related to this section:

- *Feature*: provides a high-level description of a system's functionality and groups related scenarios.

- *Language*: defines the language used in the artifacts.

  The *Gherkin* format is already used in several languages around the world. Thus, this keyword allows the tool responsible for extracting the information present in the file to identify which language was used to describe the scenarios and features of the system.

- *Scenario* or *Example*: describe a system usage scenario.

  These words are synonyms, illustrate a specific business rule and are composed of *steps*, which allows them to describe a usage scenario, as well as the expected result with the execution of that scenario. In this way, each *scenario* represents executable documentation for a system business rule.

- *Scenario Outline* or *Scenario Template*: allows the definition of usage scenarios with different *inputs.*

  These keywords are used to create *scenarios* with more than one example of *input.* Using *n* examples, the scenario will be executed *n* times, once for each *input* registered. The listing of *inputs* is done with the keyword *Examples.*

- *Background*: defines a prerequisite for the execution of the usage scenarios.

  This keyword is used to register a specific execution context within the *feature.* Thus, *background* has one or more *Given steps*, being executed once before the execution of each *scenario* registered in this *feature.*

- *Examples*: It allows the creation of a table of examples, where each column represents a variable and each row represents a value. It is a concept quite used for repetitive execution of the same *scenario*, replacing only the *inputs* of the scenario at each execution.

- *Steps*: defines the necessary steps to execute a given usage scenario. This keyword has some keywords related to it, such as *Given* to define the initial state of the scenario, *When* to define an action in the system and *Then* to describe the expected result after action. Seeking the fluidity of usage scenarios, we have the keywords *And* and *But*, which have the same meaning as the same keyword used in the past *step*.

  - *Given*:

    It is used to define the initial state of *scenario*. The purpose of its use is to leave the system in a known state for the execution of the desired test case.

  - *When*:

    It is used to describe an action on the system, which means the interaction of a third party (a user or another system) on the system under test. Generally, each *scenario* has only one *When*, defining the main action of the scenario to be tested.

  - *Then*:

    It is used to describe the *output* expected from the execution of the previous *step*. This step represents the *assert* that checks if the output obtained is equivalent to the expected output, concluding the test case.

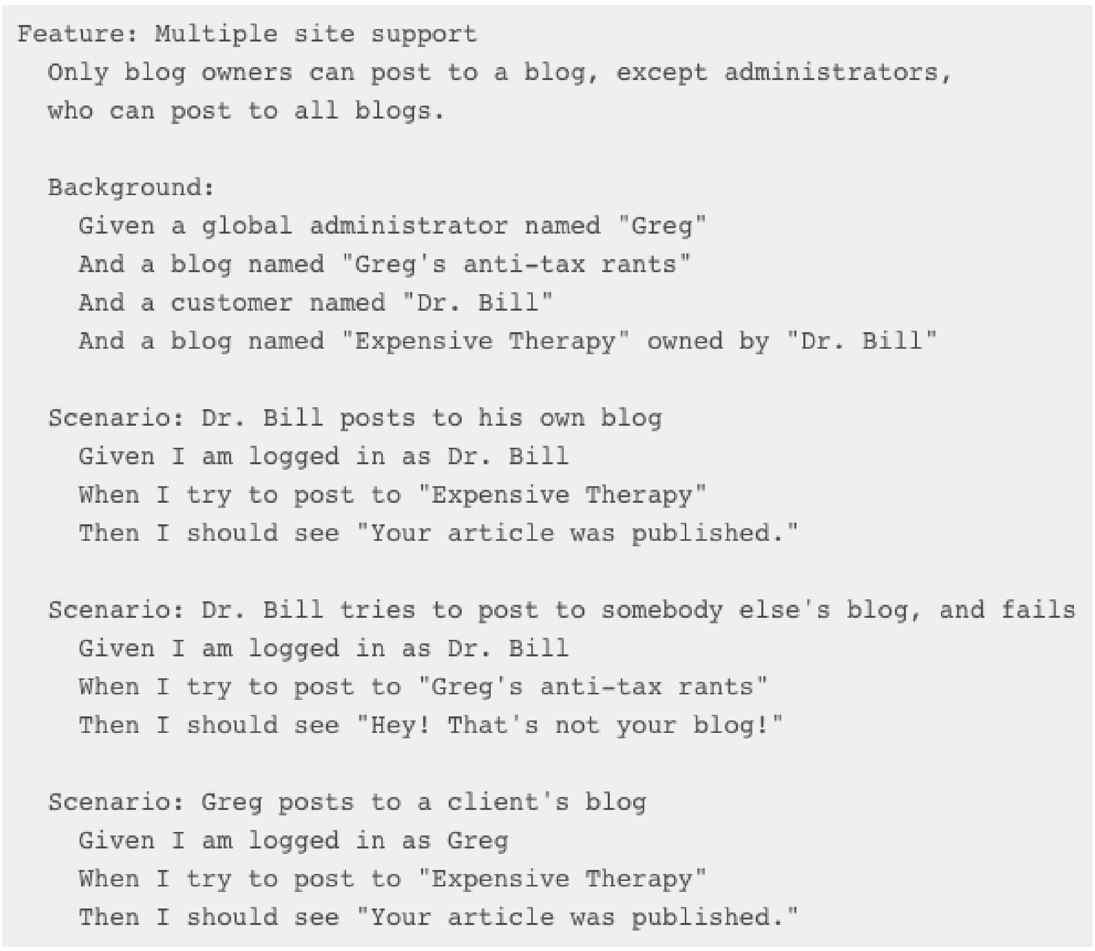
  - *And* e *But*:

    They are responsible for preventing the writing of many *Given*, *When* and *Then* and serve as synonyms for the step before them.

The Figure 2.2 shows some example of a *.feature* file, which is usually a common file that stores feature, scenarios, and feature description to be tested. The feature file is an entry point, to write the cucumber tests and used as a live document at the time of testing.

The use of the Gherkin language allows the mapping of each step to an executable test method. Thus, each scenario represents a set of test methods that will be executed in the order defined for completion of each use scenario [41]. Using this approach, feature-to-code traceability can be obtained automatically, since the system requirements become an executable and verifiable entity. However, an entity that has only a partial view of the system since the view of the system in this approach is seen as a set of isolated features.

According to [42], BDD has six general characteristics:



```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

Figure 2.2: Sample of *.feature* file [2].

- Ubiquitous Language: It is the core concept of the BDD approach. The language structure is derived from the business context, that is, it uses terms that are commonly used by stakeholders to record the behavior of the system, avoiding ambiguities.
- Iterative Decomposition Process: A collaborative and incremental process, generating features and their respective scenarios over time, with the participation of all stakeholders.
- Standardized Textual Descriptions: Defining features in a standardized manner, facilitating the understanding of what should be implemented in the system, and adding the benefit of automating the usage scenarios, making each scenario represent a set of executable test methods while maintaining an easy-to-understand structure for any stakeholder.
- Automated Acceptance Tests: The set of features and scenarios of a system, in the BDD approach, are both the system requirement and the executable test cases of

the system. In this way, an automated executable specification is obtained.

- Code-oriented Specification: Like the test, the code is also the system documentation itself, in the BDD approach. In this way, the elements of the code should represent the expected behavior, just as they should use the same terms defined according to the ubiquitous language of the project.
- Influence in several stages of development: The BDD approach is present in several stages of development, such as in the planning phase (business behavior), analysis (definition of features and scenarios), implementation and evolution of software. Thus, the developer is bound to think about the behavior of the component being developed.

Several tools and frameworks support the BDD approach for the most varied technologies and contexts. For the realization of this research, the main technologies to support the BDD approach were raised, as can be seen in Table ref tab: frameworks. To identify the relevance of the tool in the development community, the quantities of *stars* and *forks* were analyzed in their GitHub repositories. It is worth mentioning that in cases where the tool is distributed in more than one repository, its numbering of *stars* and *forks* have been added, in order to obtain the complete view of the tool, including all the modules.

Tool	**Supported Language**	**Stars**	**Forks**
SpecFlow[2]	.NET	1330	580
Jasmine[3]	JavaScript, Python, Ruby	14770	2610
Concordion[4]	Java	158	60
Behat[5]	PHP	3260	963
Behave[6]	Python	1571	418
Lettuce[7]	Python	1161	314
Codeception[8]	PHP, JS	5600	1676
Jbehave[9]	Java	256	247
Cucumber[10]	Java, Ruby, Python, JS	13339	4184

Table 2.1: Frameworks that support the BDD approach.

[2]https://github.com/techtalk/SpecFlow
[3]https://github.com/jasmine
[4]https://github.com/concordion
[5]https://github.com/Behat
[6]https://github.com/behave
[7]https://github.com/gabrielfalcao/lettuce

Table 2.1 highlights the prominence of the Jasmine and Cucumber frameworks, being the most recognized and used by the development community. The framework used throughout this work for automation and standardization of BDD features will be Cucumber. However, any other framework can be used to apply the approach presented in Chapter 3.

According to [32], the BDD approach enables the development team to focus their efforts on the identification, understanding, and construction of features of business importance, allowing these resources to be well designed and well implemented. Although it is an apparently more dynamic way of writing about requirements, according to so-and-so, the writing of BDD scenarios does not follow a pattern, or rule to be applied.

Due to a lack of formalization of the definition of features in the format of BDD scenarios, some problems may arise, such as the problem of incomplete or inconsistent requirements. Therefore, according to [43], it is important to structure knowledge about BDDs in the form of quality attributes and use a question-based checklist in the form of a document accessible to software teams.

2.4 Program Spectrum

The Program Spectrum concept refers to the system execution path, recording the points of the code that occur during the execution of a particular feature [44]. This information can be useful for comparing the behavior between different versions of the same system, making it easier to identify the fault that generated the failure.

Program Spectrum provides a behavioral signature for each execution, allowing internal knowledge of the implementation of each system usage scenario. The Program Spectrum is obtained from the instrumentation of the code, registering the execution path of the same. In this way, it is possible to obtain the behavior of the system according to each feature/input pair executed. This information can be useful during debugging and software testing activities. In relation to the test activity, [37] proposes a new white-box approach, called *I/B Testing*, or Input/Behavior Testing, where path-spectrum is used to identify points of failure, even if the output appears to be correct.

The concept of Program Spectrum can be classified as follows: [44]:

- Branch spectra: Records the set of conditional branches exercised during system execution. Depending on its purpose, it is called Branch-hit Spectrum (BHS), or Branch-count Spectrum (BCS). When the purpose is only to find out whether a

[8]https://github.com/Codeception
[9]https://github.com/jbehave
[10]https://github.com/cucumber

particular entity has been executed or not, it is the technique **BHS**. If the goal is to count how many times each entity has been executed, this is the **BCS**.

- Path spectra: Records the set of intraprocedural and loop-free paths exercised during system execution. As in the previous concept, it can be classified into two categories: (1) Path-Hit Spectrum (PHS) and (2) Path-Count Spectrum (PCS), which make it possible to verify if a specific path was exercised and how many times each path was executed, respectively.
- Complete-path Spectrum (CPS): Determines the complete path of system execution, taking into account all lines of code exercised.
- Data-dependence Spectra: Determines the pairs of [statement/use] exercised during system execution. This concept can also vary, being classified in Data-Dependence-Hit Spectrum (DHS), which is concerned only with recording whether a particular pair was exercised or not, and Data Dependence-Count Spectrum (DCS), which is concerned with recording how many times each pair was exercised during the execution of the system.
- Output spectrum (OPS): Records the output produced from the system execution.
- Execution-trace spectrum (ETS) Records the sequence of program statements exercised during the execution of the system.

In Figure 2.3, a representation of the Complete-path Spectrum is presented, where the entire execution path of Listing 2.2 is registered.

```
void print_average(){
   num_inputs = read_num_inputs();
   i = 0;
   while (i <= num_inputs){
      value = get_input();
      sum = sum + value;
      i = i+1;
   }
   average = compute_average(num_inputs, sum);
   printf(average);
}

```

Listing 2.2: Sample code.

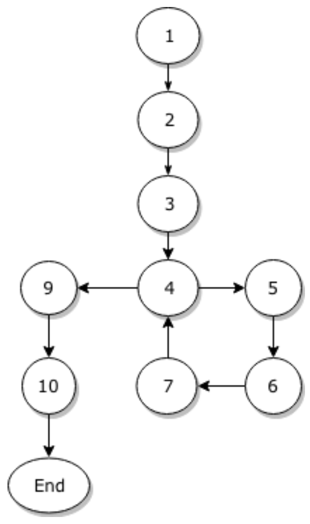


Figure 2.3: Execution path.

The use of the Program Spectrum concept may differ significantly, as the classifications presented above can be applied for different purposes, as could be observed in the work of [17], where *branches*, *functions* and *statements* were analyzed with the objective of maximizing the efficiency of the prioritization of test cases based on the Operational Profile.

Chapter 3

The Feature-Trace Approach

Our approach, *Feature-trace*, aims to define a method for the semi-automated generation of the OP and to allow for the prioritization of software testing by means of BDD artifacts and quantitative SUT information. Our approach uses BDD artifacts to enable the generation of the OP, while minimizing the usual effort involved in the OP synthesis process. As such, *Feature-Trace* allows for the seamless analysis of the distribution of the test effort employed throughout the source code and the prioritization and selection of test cases based on OP, program spectrum, number of impacted features, and the methods complexity.

An overview of the *Feature-Trace* approach is depicted in Figure 3.1. It consists of three main modules: *Automated Runtime Module* (ARM), *Analyzer*, and *Visualizer*.

Firstly the ARM verifies the language of the SUT, analyzing the feasibility of the analysis according to the units of extraction available. After this analysis, the SUT goes through a static analysis process where all the entities of the software from BDD artifacts to code entities are obtained. By these means, the SUT is instrumented to enable the execution traces to be obtained. Once the ARM has processed the information from the execution traces, they are persisted in a database.

The *Analyzer* module, in its turn, is responsible for: (1) generating the map of features, which highlights the traceability between the requirements and the code of the SUT; (2) obtaining and distributing the probabilities of occurrence of each SUT entity, achieving the OP; and (3) presenting data from each entity according to the spectrum count, the number of *impacted features* (IF) and the analysis of the complexity of each method.

Finally, the *Visualizer* module is responsible for the interface between the approach and the user, presenting the global view of the SUT and the quantitative data of each entity, besides guiding the prioritization and selection of test cases focusing on the growth of software reliability.

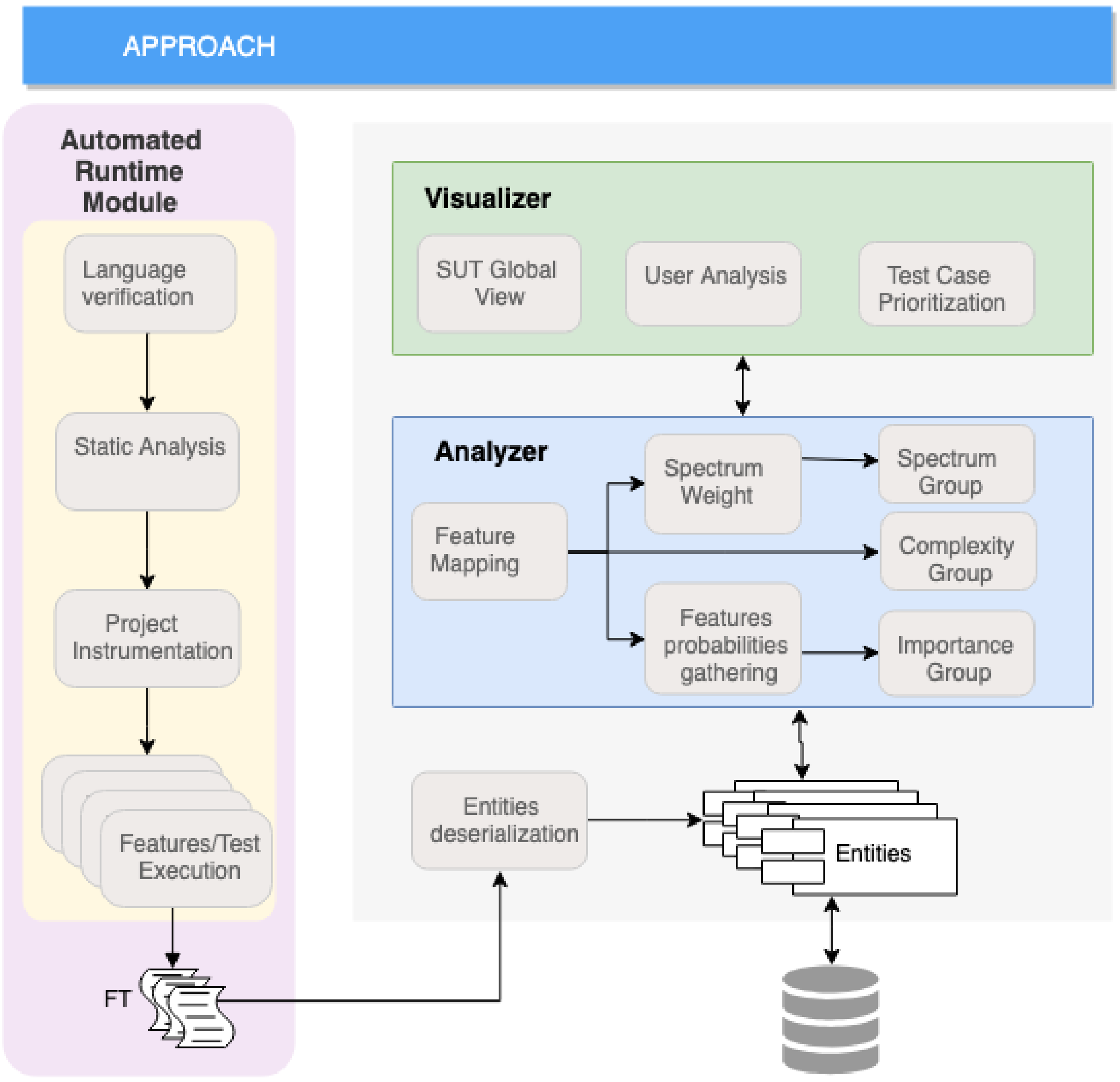


Figure 3.1: Feature-Trace Architecture.

The following sections provide a detailed explanation of the main modules that constitute our Feature-Trace approach.

3.1 The Automated Runtime Module

The main objective of the ARM is to extract information from the SUT, statically and dynamically, persisting the information required to carry out the analysis present in the other two modules: Analyser and Visualizer. The main activity of the ARM is to extract the entities needed to generate the OP, both from the BDD artifacts and the source code. This activity is done in "Feature/Test Execution" activity, presented in Figure 3.1.

As shown in Figure 3.1, the first task performed by ARM is to verify the language of the SUT, since static and dynamic analysis depends directly on the technology used by SUT. Technology verification takes into account some patterns of each language, such as the format of present files and the presence of configuration files (such as ruby's *gemfile*).

After language identification, the SUT static analysis is performed, extracting all classes and methods present in the project, all test cases, and all BDD scenarios. Static analysis also seeks to extract some code complexity metrics, such as Cyclomatic Complexity, ABC Score, and Number of Code Lines of each method. In this work, complexity metrics are extracted using *Excellent*[1] as a supporting tool. This tool was chosen because of its ease of integration as it is a CLI (Command Line Interface) based tool. This way, the tool developed in this work, *Trace Feature*, can execute the *Excellent* and obtain the result of the execution, recording the necessary metrics for each of the existing methods in the project. By extracting all methods, tests, scenarios, and complexity metrics, all data is sent to the server for storage in the database for future analysis (Analyzer and Visualizer modules).

At the end of the static analysis, some source code instrumentation activities are required to obtain traceability data by identifying the set of methods executed from a given scenario or test case. In this first version presented in this paper, the instrumentation process is done to configure *simplecov*[2]. For this, it must be inserted in the *gemfile* (if it does not exist), and the directories to be analyzed must be defined. After the correct configuration, the environment is prepared for project execution in the next step (dynamic analysis). All of these activities are fully automated using the Feature-Trace tool presented in this paper, which makes the analysis process simpler and faster.

By extracting all necessary static information and setting up the environment, the test case execution process begins to identify the methods exercised by each test case or BDD scenario. At the end of the execution of each test case, the set of executed methods is sent to the server to perform feature mapping, building a graph that joins each feature to each scenario and each scenario to all the methods exercised by it. Mapping also involves the relationship between existing methods and the test cases (unitary and integration) that exercise them. Thus, you can map all entities involved in the project, from the features and scenarios (requirements), the methods (source code), and the test cases that verify these methods. This set of related entities enables the analysis of various data following various strategies, as will be presented throughout the work.

When referring to the extraction of BDD features/scenarios, we should highlight the strategy used for this, since this work seeks to relate the entities of the approach BDD

[1]https://github.com/simplabs/excellent
[2]https://github.com/colszowka/simplecov

with the entities present in the definition of the Operational Profile. The following is the strategy used to perform this entity relationship.

3.1.1 Operational Profile Entities

Regarding the OP entities extraction, we start from the highest level of the OP, the **Customer Profile**, that refers to the customer who purchases the software, grouping types of users into a wider classification [1]. We should note that the BDD approach does not directly support the definition of an entity related to it. However, BDD allows the inclusion of user-defined "tags" to group features into similar groups. This capability can be used to define the **Customer Profile**. The same strategy can be applied at the **User Profile** and **System-mode Profile** levels, as they do not have a corresponding BDD entity.

In relation to the **Functional Profile** level, the functionalities are defined in the software user view. This entity can be related directly to the concept of BDD Feature, which also represents the functionality of the software in the view of the user [25].

After identifying the **functions**, the **key input variables**, which define the variations of the use of each **function**, must be raised [1]. In the BDD approach, variations of a feature are defined from the writing of usage scenarios. By these means, **key input variables** can be obtained directly from the concept of BDD scenarios.

Finally, at the last level is the **Operational Profile**, which is obtained from the execution trace analysis of each **key input variable**. That is, operations are mapped into entities of the SUT source code enabling the execution of specific functionality. In this work, operations are mapped in software methods, but the degree of granularity must be adapted according to the OP analysis need.

Table 3.1 presents the proposed relationship between the entities of the OP and the entities of the BDD approach.

Table 3.1: Operational Profile from BDD artifacts.

OP Entities	**BDD**
Customer	User-defined tags
User	User-defined tags
System mode	User-defined tags
Function	*Feature*
Key Input Variable	*Scenario*
Operation	*Trace*

Following these relationships, the ARM uses automation strategies to obtain all the entities of the OP. The mapping between the OP entities and the BDD approach makes it possible to obtain some vital information, as one can execute BDD scenarios and unit/integration tests in such a way that execution is tracked and analyzed. As suggested by the *Covrel* approach [17], the OP (black-box) information should be weighted with the *Program Spectrum* (white-box), maximizing the final reliability of the software, since, from this weighting, we avoid the stability of the reliability growth of the SUT after few test cycles were performed. So the ARM extracts the Program Spectrum of all SUT entities. In addition to this information, we obtain, statically, information about the complexity of each method, such as *ABC score* [45], and *Cyclomatic Complexity* [46]. In addition, the ARM obtains the result of each test case, recording whether the test has passed or failed, allowing an analysis of the failure history.

3.2 The Analyser Module

This module is responsible for constructing the mapping between the entities of the software, relating from the requirements to the computational units that compose them. The output of the information provided by the ARM is incrementally submitted to the Analyzer Module, where each feature is executed individually. By these means, the Analyzer Module receives requests sequences containing a feature with its scenarios and their respective traces of execution. We should note that in the Feature-Trace approach, the process of mapping SUT entities occurs in an automated way in the Analyser Module.

We explain the process of the Analyser Module through the Algorithm 1. The Algorithm receives a tuple that contains the feature and scenarios with their respective execution traces and it must return success or fail. In line 2, the Algorithm instantiates the feature in a python object. As the process of obtaining the information occurs incrementally, the database must guarantee there is no redundant data, as it can be observed in lines 3 and 10 of Algorithm 1. Lines 4, 11 and 17 verify the existence of the object in the database, for updating. The relationship between Features, Scenarios, and Methods is performed according to lines 13 and 18, arranging in such a way that a feature has one or many scenarios and a scenario has one or many methods executed. In line 16 the set of persisted methods of the SUT, extracted and registered through the ARM, is then filtered to relate with the current scenario. This filter facilitates the verification of the completeness of the SUT since it may output those methods that were neither executed by BDD scenarios nor by unit/integration tests.

At the end of Algorithm 1, all entities involved in the process of feature analysis must be registered in the database with their appropriate relationships. Figure 3.2 illustrates the relationship between features, scenarios, and methods, where the features are on the top in blue color, the scenarios are in the middle in yellow color and the methods are on the

Algorithm 1: Feature Mapping Algorithm.

```
Input: String f_trace
Result: Boolean
Function create_entities(f_trace)
    feature = instantiate_feature(f_trace[‘feature’]);
    f_base = verify_feature_base(feature);
    if not f_base then
        feature.save();
    end
    var scenarios = f_trace[‘scenarios’];
    for scenarios as scenario do
        var sc = instantiate_scenario(scenario);
        var base_sc = verify_scenario_base(sc);
        if not base_sc then
            sc.save();
            sc.feature = feature;
        end
        for scenario[‘methods’] as method do
            met = Method.filter(method_id=method[’method_id’]);
            if met then
                sc.executed_methods.add(met[0]);
            end
        end
    end
    return True
```

bottom of the Figure in green color. We should note that, from this structure, the upper levels of the OP regarding customer, user and system mode may be also complemented if the set of *User-Defined Tags* are specified and then previously analysed by ARM.

In addition, a procedure analogous to the one presented in Algorithm 1 is performed in the Analyser Module to extract the methods that are invoked in the execution of unit and integration test cases. This information is possible given that the ARM also supports unit and integration test execution.

In order to conclude the steps of OP generation, the probability of occurrence of each mapped entity must be performed. As in traditional OP generation approaches, these probabilities must be obtained from the domain context. Such information can be obtained from interaction with domain experts or experienced users [1]. Thus, high-level entities can receive estimated probability information from the highest OP level to the level of **key input variables**. With this information, the software execution trace enables the distribution of probabilities between the levels of **Functional Profile** and **Operational Profile**, completing the OP. We should note that the probability of

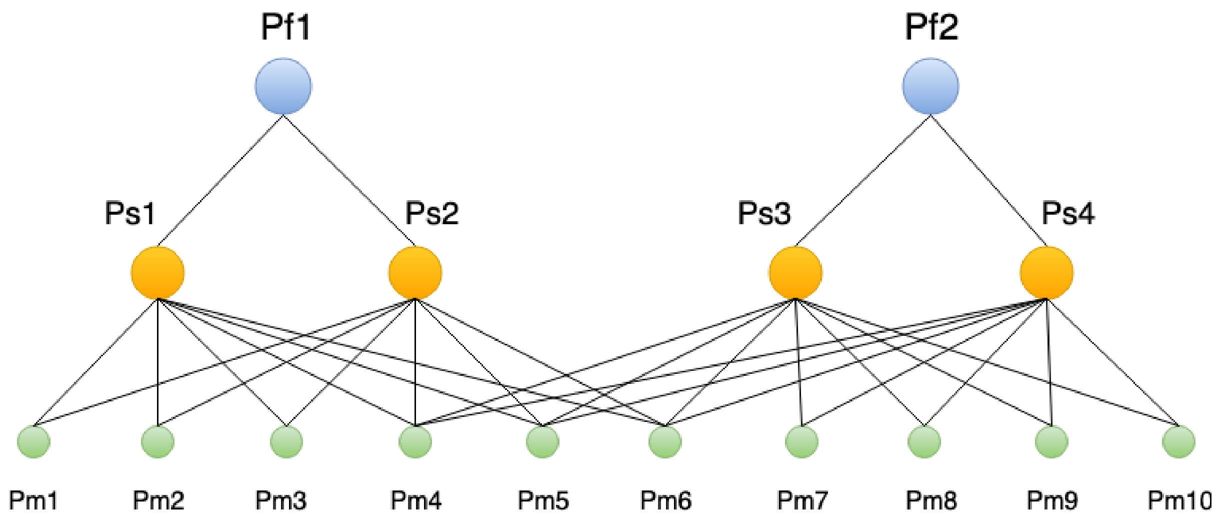


Figure 3.2: Map of *Features*, *Scenarios* and Methods.

occurrence of the method is simply derived from the sum of the probabilities of occurrence of their corresponding invoking scenario.

After generating the OP, the Analyzer module distributes the analyzed methods into three groups, where each group is further classified into *High, Medium and Low* level. Those groups take into account different criteria, allowing for comparisons and analyses that assist the prioritization of test cases. The first group, called the *Spectrum Group*, is based on the number of test cases that exercise each method. The second group, the *Complexity Group*, takes into account some metrics of complexity and the third group, the *Importance Group*, uses the OP information. The distribution between the levels of each group can be done via clustering or any grouping strategy. Then, the use of these metrics facilitates the prioritization and selection process by the testing team based on (1) the SUT reliability impact (importance group), (2) the amount of test-cases that exercise the entity (spectrum group), and (3) the methods complexity (complexity group). The 2 Algorithm provides a simple example of how importance grouping (Operational Profile) can be done.

Other relevant information that should be taken into account during the test case prioritization or selection process is the number of *Impacted Features* (IF) by a given code entity such as methods or classes. This information is straightforward since the entities have already been mapped. In this way, the Analyzer module computes and records this information following the one presented in Algorithm 3. To obtain the number of features impacted by a given method, we observe all the scenarios that execute the method and include their features in a variable of the type *Set*[3], as shown in line 4. This strategy avoids the repetition of features since some scenarios that execute the method are related

[3]Set elements are unique. Duplicate elements are not allowed.

Algorithm 2: Groups Distribution.

```
   Input: List methods
   Result: Dict levels
1  Function distribute_importance_group(methods)
2      var levels = {
3          "high": [],
4          "medium": [],
5          "low": []
6      }
7      for methods as method do
8          if method.get_probability() < low then
9              groups["low"].append(method);
10         else
11             if method.get_probability() < medium then
12                 groups["medium"].append(method);
13             else
14                 groups["high"].append(method);
15             end
16         end
17     end
18     return groups;
```

to the same feature. The resulting set, after line 5, has all the features impacted by the analyzed method.

Algorithm 3: Count Impacted Features.

```
   Input: Method method
   Result: Method method
1  Function count_impacted_features(method)
2      features = set();
3      for method.scenarios as scenario do
4          features.add(scenario.feature);
5      end
6      return len(features);
```

The distribution of probabilities is based on the edges of the entities graph. A simple and practical example is presented in the 4 Algorithm. Since a method is related to specific BDD scenarios and these scenarios have a probability of occurrence derived from business area analyses, customer interviews, or any other strategy, it is possible to distribute this probability. To do it, add the likelihood of occurrence of all scenarios that exercise a given method, as presented in line 4 of the Algorithm.

Algorithm 4: Method probability (OP).

```
Input: Method method
Result: Method method
1 Function get_probability(method)
2     method.probability = 0;
3     for method.scenarios as scenario do
4         method.probability += scenario.probability;
5     end
6     return method
```

In this way, the Analyzer Module provides the information obtained and the analysis performed to the Visualizer Module so that those responsible for the testing process adopt the strategy they want to distribute the testing effort and to prioritize the test cases.

3.3 The Visualizer Module

As shown in Figure 3.1, the Visualizer module enables: (1) the global view of the SUT, making evident its traceability; (2) performing analyzes from the data crossing and (3) using the generated data to support the test case prioritization or selection according to the desired strategy. This Module is responsible for presenting the outcome from the Analyzer Module. Among the information presented, we highlight the groups of entities based on criteria such as complexity, Program Spectrum and OP.

In addition, the Visualizer Module enables a global view of the software from the perspective of requirements, code and test entities in a graph-like structure, as depicted in Figure 4.1. This representation provides a view of the reification of requirements into source code, as well as makes clear the proximity of methods, that is, methods that make up the same features, always occurring together, and specific methods, composing only one or few features.

The global view of the software automates the process of identifying the requirements in the source code, which is recognized as an essential activity for the software maintenance and evolution process [47]. In this way, the Visualizer Module enables the testing team to graphically analyze the traceability of the software project, as well as to analyze the quantitative data of the traced software entities and thus guide its maintenance, evolution and testing process.

This set of visualizations and information of the software entities enables the application of several techniques for prioritizing test cases, from black- to white-box techniques. From our Feature-Trace approach, it is possible to apply several of the techniques presented in [12] and [15]. For example, among the white-box prioritization techniques, the

Feature-Trace approach supports techniques based on both *Total Coverage*, where priority is given to test cases that cover the largest number of source code units, as well as *Additional Coverage*, which seeks to select the test cases that cover the largest number of entities not yet covered by the set of test cases executed [48]. The approach also allows for the use of prioritization techniques that are based on the diversity of the generated trace, using, for example, Global Maximum Distances algorithms [49]. In relation to black-box testing prioritization techniques, the Feature-Trace approach allows the use of techniques based on the requirements [50] and using concepts such as OP [1]. Finally, from the logs of the testing activity, our approach may also allow the application of techniques that use the history of test case failures as a prioritization or selection criteria.

3.4 Development Architecture

The Feature-Trace tool is in its first version, at the time of completion of this work and, currently, it is divided into two major parts: a client (Automated Runtime Module) and a server, containing the Analyzer Module and Visualizer Module. Both parts were developed in Python 3, with ARM made with pure Python and the server using the Django framework[4].

3.4.1 Automated Runtime Module

Regarding ARM, its implementation follows that shown in Figures 3.3 and 3.4.

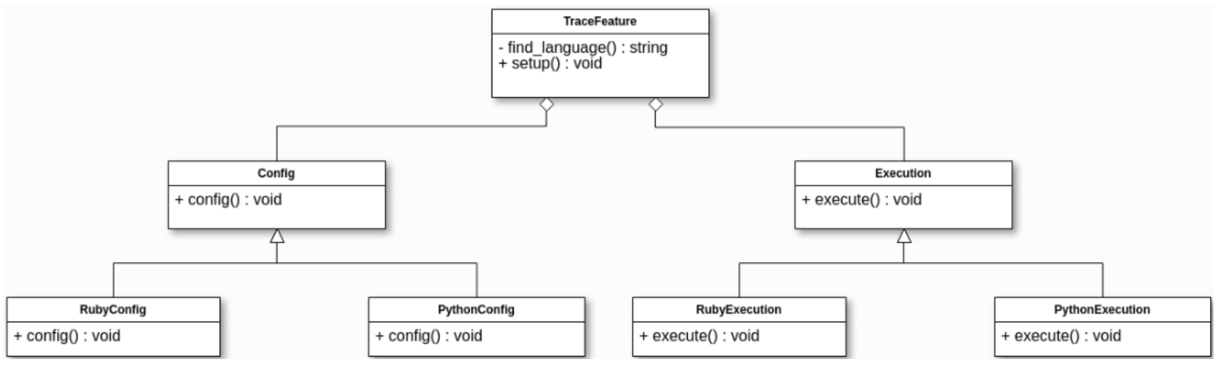


Figure 3.3: ARM Architecture.

In ARM, the starting point of execution occurs in the class *Feature-Trace*, being responsible for defining the user interface (CLI) and distributing the activities to *Config* and a *Execution*. So far, the proposed tool is only capable of analyzing projects developed in Ruby. However, its architecture has been designed in such a way that it makes evolution

[4]https://www.djangoproject.com/

easy, including new capabilities (other languages and technologies). In this sense, the ARM module is divided into the Configuration and Execution parts of the SUT. Each of these parts is defined from an abstract class that defines the methods necessary to support the analysis of a given language. That is, to insert a new capacity, just insert a class in the "RubyConfig" or "PythonConfig" pattern, extending from *Config*, and a class in the "RubyExecution" or "PythonExecution", extending from *Execution*.

The child classes of *Config* are responsible for configuring the project to be analyzed, making it possible to extract the necessary information. To do this, you must first install and set the gem "Excellent" (in the case of Ruby), to obtain the complexity metrics for each method. Also, *Simplecov* must be installed and configured to identify the lines executed by each scenario or test case, defining the set of directories to be analyzed by the tool. In addition, the tool must install the analyzed project, allowing its local execution, given that all test cases must be executed.

The *Execution* child classes are responsible for executing the extraction tool of complexity metrics for each method and, then, the execution of all present test cases. Execution is primarily responsible for the functioning of the ARM, also communicating with the server to record all data extracted locally.

The execution of test cases and information extraction depends on the use of some support classes, which generate the initial relationship structure between the entities. This relationship can be seen in Figure 3.4. We note that a Feature relates to a set of Scenarios, which can be a *SimpleScenario* or a *ScenarioOutline*. A *ScenarioOutline* refers to BDD scenarios that have sample tables, in which each row of the table can be considered a new scenario (new parameters). Each Scenario is related to a set of n methods, which are exercised during the execution of the Scenario. Also, each Test Case is related to a set of methods that are exercised from its execution.

From the instantiation of this whole set of entities, the ARM module can send all the information obtained to the server for storage and future analysis in the next two modules of the approach.

Figure 3.5 shows the folder structure used in the ARM source code. It shows that the tool has a *core*, where the model classes presented in Figure 3.4 and the two abstract classes that define the configuration and execution of the test cases are defined. Within *core*, you will also find folders for each of the languages supported by the tool. Currently, only Ruby is supported, however with the evolution of the tool, new folders should appear, such as " Python ", " Java ", etc.

This module is packaged using the *pip* [5] standard, facilitating the installation process. All the functionality of the module can be seen from the *–help* flag, which are:

[5] https://pypi.org/project/pip/

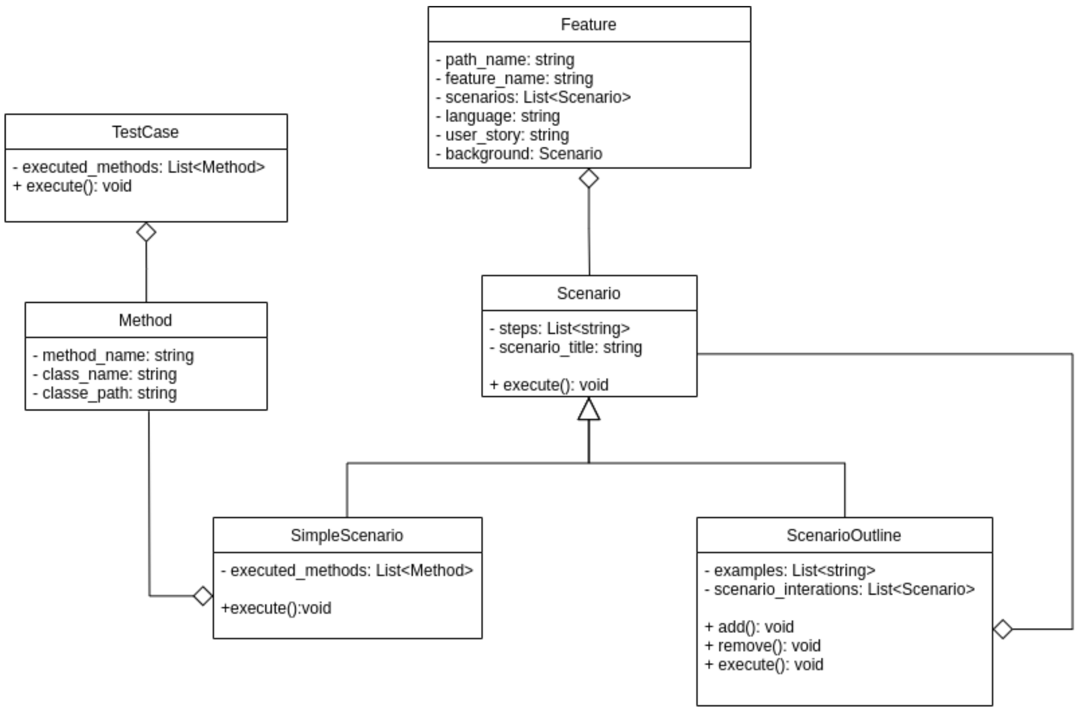
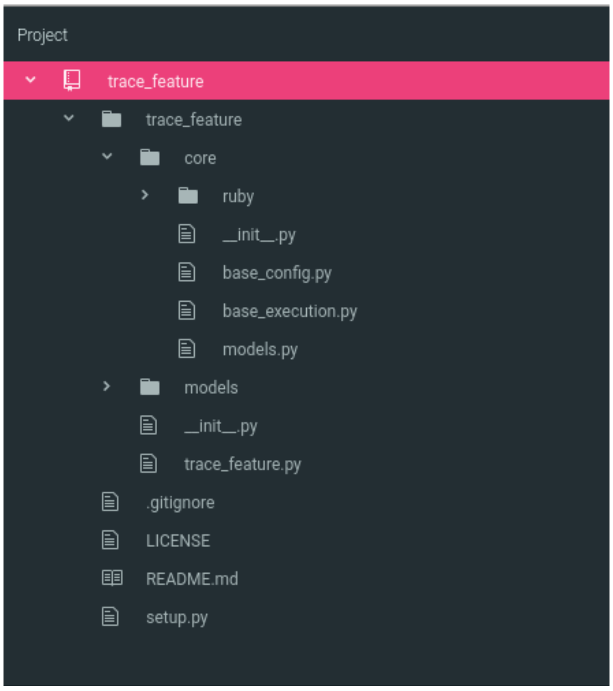


Figure 3.4: ARM Models Architecture.

- *-s, –scenario* (INTEGER): This is the scenario's corresponding line that can be found at the feature file. This command execute that scenario and send information to the server.
- *-f, –feature* (TEXT): This is the file's name where the feature is defined. This command execute that feature and send information to the server.
- *-p, –project* (TEXT): Flag used to detail the name of the project to be analyzed. The default is the current directory.
- *-l, –list*: This option list all features and scenarios into this project.
- *-t, –spec*: This option execute spec files tests into this project and send information to the server.
- *-m, –methods*: This option read all methods into this project.
- *-a, –analyse*: This option analyse all methods into this project, extracting the complexity metrics.

In addition to the possibilities presented above, the execution without any *flag* makes the process run as a whole, doing the static analysis of the entire project and then executing all the features and all the test cases, sending all the information to the server. The project structure is divided into two main layers: the most abstract layer, defined as "base", in which the classes "base_config" and "base_execution" are defined to standardize the methods necessary for future extensions. From this layer, the lower layer should extend from the upper layer, creating classes of "config" and "execution" according to the characteristics of each new language to be analyzed. The "ruby" directory is one of the possible directories where child classes are defined.



```
Project
trace_feature
    trace_feature
        core
            ruby
            __init__.py
            base_config.py
            base_execution.py
            models.py
        models
        __init__.py
        trace_feature.py
    .gitignore
    LICENSE
    README.md
    setup.py
```

Figure 3.5: ARM Folders.

3.4.2 Analyser and Visualizer Modules

The basic architecture of the server follows the Django standard, that is, Model-View-Template (MVT), where Model is responsible for defining the models, View is responsible for controlling and managing the functionalities (Controller), and Template is responsible for defining the interface with system users. The basic structure of server models also uses the same set of model classes as ARM, shown in Figure 3.4. The main difference is related to the record in the database with all defined relationships. After this registration, all analyzes performed are based on specific *queries*, such as those presented in Algorithms 1, 2 and 3.

In addition, a package for *covrel* was included, where some analyzes suggested by *covrel* [17] are performed. Among them, the distribution of probabilities of occurrence among the entities stands out, the Spectrum count, and the distribution of the entities in groups of importance. Unlike the *covrel* approach, in this research, three classes of groups are defined for the distribution of entities, namely: Importance Group, Spectrum Group, and Complexity Group.

To present the graphs that represent the relationships between all the entities of the project (Global view of the software), the tool *D3.js* footnote https://d3js.org/ was used, generating images such as those presented in 4.2 and 4.1.

3.5 Costs Involved

Regarding the costs involved with the use of the tool proposed in this work, the costs related to the execution of the whole set of test cases stand out, being this the main bottleneck of the tool execution. However, this cost, although important, could not be different, since to obtain the data extracted at run time, it is necessary to run the software. In addition to the tests execution cost, the cost related to software instrumentation is added, given that the coverage data is being collected at all times. This cost is equivalent to the cost generated by using the *simplecov* tool, known to the ruby community.

When executing each test case, the coverage data is generated by *simplecov*, and these are processed by the Feature-Trace (ARM) tool and sent to the server, being a request for each *feature* or test case (integration/unitary). The processing of this data is nothing more than the construction of the JSON necessary to power the server, generating low cost for the machine. From this stage, all processing takes place on the server, avoiding wear and tear on the part of the client.

Chapter 4

Case Study

We evaluate our Feature-Trace approach to its ability to analyze different metrics related to the software testing process, and thereby enable the distribution of the test effort throughout the source code. This yields the following research question:

> Can BDD artifacts be used to generate the OP and guide a prioritization and selection testing process?

Our approach proposes a method that enables the prioritization and selection of test cases from the extraction and analysis of metrics obtained directly or indirectly from BDD artifacts and source code. The metrics analyzed are (1) the **Operational Profile**, obtained from the mapping of BDD-related entities, (2) the **Program Spectrum** related to BDD Features and to the Unit and Integration test cases, and (3) the **complexity** of the method taking into account the Cyclomatic Complexity, the ABC Score and the number of lines (LOC) of the method. In order to automate our approach, we implemented the *Feature-Trace* tool, developed in Python and available as a GitHub project[1].

The evaluation of the approach and, consequently, of the support tool, was made from two steps. The first step aims to analyze the ability of the support tool to extract the necessary information and present it to the user objectively and simply. For this, data was extracted in a large and complex system, which will be presented below. The second validation step seeks to obtain feedback from people who have tried using the approach and tool proposed in this work. Both validation steps will be described and discussed throughout this Chapter.

[1]https://github.com/BDD-OperationalProfile

4.1 Evaluation Step 1

4.1.1 Systems Under Test

We carried out a case study in **Diaspora**: a social network highly acknowledged in the open source software community. It has 12,226 *stars* and 2,915 *forks* in its repository in *Github*[2]. It is implemented in Ruby on Rails and has a total of 72,674 lines of code, out of which 40,546 lines of Ruby code with 1,571 methods, and a total of 2,975 lines of *Gherkin* code, distributed throughout 68 BDD features. It also has a set of 2679 unit and integration test cases.

4.1.2 Experimental Setup

To analyze the Feature-Trace approach some tools and configurations have been adopted. In order to obtain the execution trace of the scenarios and unit/integration tests, we use the *Simplecov*[3] tool, making it possible to extract the trace of each software execution in the JSON format, facilitating the reading by the Feature-Trace tool for traces analysis. Note that the *Simplecov* tool provides a JSON file containing the file names and line numbers with their occurrence information.

The initial process of the analysis occurs with the static analysis of the SUT source code. This analysis is done in an automated way by the Feature-Trace tool. Some information is obtained from the use of the *Excellent*[4] tool, which makes it possible to obtain the Cyclomatic Complexity, ABC Score and Number of lines of the method. Such automation has also been integrated into the Feature-Trace tool. The execution of the BDD scenarios is done with the support of the *Cucumber*[5] tool, being executed for each scenario present and also integrated into our Feature-Trace tool.

4.1.3 Results and Analysis

The application of the Feature-Trace approach in the Diaspora project made it possible to analyze the ability of Feature-Trace to extract relevant information from software projects using the BDD approach.

The extraction of information made it possible to identify the distribution of the testing effort taking into account several questions, as presented in the Table 4.1. We should note that, for the sake of space, it is not possible to list all methods of Diaspora

[2]https://github.com/diaspora/diaspora
[3]https://github.com/colszowka/simplecov
[4]https://github.com/simplabs/excellent
[5]https://cucumber.io/

in this work. We provide the full list of Diaspora software methods analysis, as well as all the artifacts involved in this case study, at Github[6].

Table 4.1 shows that we obtain the OP, the number of IF (Impacted Features) and the number of test cases that exercise each SUT method. We should note that, although Feature-Trace also gathers static information regarding the software complexity, such as Cyclomatic Complexity, ABC Score and Number of Lines, this information has been removed from Table 4.1 to highlight those information at stake as an outcome of our approach. Also, we selected one method from each importance group (based on the OP) as an example.

Based on the outcome of Feature-Trace on Table 4.1 for the Diaspora software, we should prioritize more test cases to method M1 (which has the highest OP), or we should focus on method M2 which contains few test cases and a considerably higher number of IF compared to method M3?

In addition to the information presented in Table 4.1, the presented methods M1, M2 and M3 also have information regarding the testing effort of these methods based on cyclomatic complexity, ABC score, and LOC. That is, the testing team can analyze the situation and decide which unit of code should be tested in the next testing cycle, for example, the unit that has the highest OP and lowest Spectrum? But what if we take into account the effort of the team in conducting this test? Is this order maintained? Discussions like this are very important to the software testing process, highlighting one of the benefits promoted by our Feature-Trace approach.

Table 4.1: Methods information obtained with Feature-Trace.

Method	**Occ. Prob.**	**Imp. Feat.**	**Spec.**
person_image_link PeopleHelper (M1)	*88.4785%*	61	2679
current_user_person_contact PersonPresenter (M2)	*53.8023%*	45	92
3. extension_whitelist UnprocessedImage (M3)	*16.3767%*	*14*	116

To analyze the generation of the OP presented in Section 3.2, we take into account two user profiles of Diaspora software, which we call UP1 (User Profile 1) and UP2 (User Profile 2). The probability of occurrence of the features and scenarios of both hypothetical profiles were generated taking into account a profile of a typical user of the social network, which uses it as an entertainment medium (UP1), and a profile of a user that uses the social network as a Digital Influencer (UP2). Table 4.2 presents some information about the OP generated for the Diaspora software. Among them, there are the features and

[6]https://github.com/FeatureTrace/SBES2019

scenarios most and least executed by each User Profile, as well as the most executed method by both UP1 and UP2. It can be noticed that for both user profiles, the most executed method is *"person_image_link"*. It is due to the fact that the Diaspora software is a social network that needs to render the user's photo in several scenarios of usage. For better visualization, we choose among those methods that do not occur during the execution of all the features and scenarios of the SUT. We postulate that the methods common to all features and scenarios (107 methods) are related to the pre-configuration required by all features executions. This set of methods should be analyzed in the next works.

Table 4.2: Example of the OP distribution.

Users Profiles	UP1	UP2
Feature with Highest OP	*Preview Posts in the Stream (2.9372%)*	The public stream (2.6668%)
Feature with Lowest OP	*Auto follow back a user (0.0559%)*	Managing authorized applications (0.0004%)
Scenario with Highest OP	*preview a post on tag page (0.6590%)*	*seeing public posts as a logged out user (1.5330)*
Scenario with Lowest OP	*preview a photo with text (0.1871%)*	*revoke an authorization (0.0002%)*
Top OP Method	*person_image_link PeopleHelper (83.7784%)*	*person_image_link PeopleHelper (84.6296%)*

Regarding the global view of the SUT, Figure 4.1 shows the representation of the *"Preview Posts in the Stream"* feature (in orange), as the main feature of UP1, its scenarios (in purple), and corresponding invoked methods (in green). In a top-down view, the image allows a broad view of the set of methods that are involved with all scenarios (automatically positioned at the center of the graph). Therefore, these are the essential methods for running the Feature as a whole. It is also important to highlight the specific methods of each feature usage scenario (prominently positioned). This graph-view perspective represents the mapping of entities of the Diaspora feature *"Preview Posts in the Stream"* and makes evident the software traceability, allowing the knowledge of the software product as a whole into various abstraction levels.

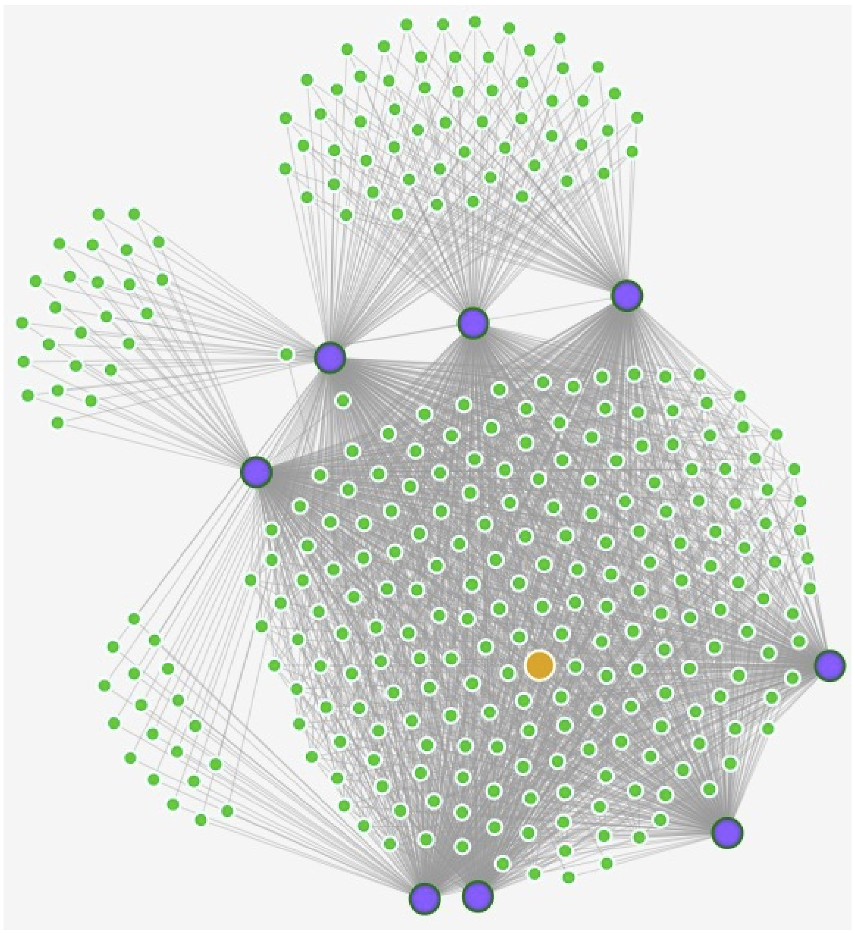


Figure 4.1: Global view of the feature *“preview posts in the stream”*.

In addition to the top-down visualization, the Visualizer Module made possible the representation of a bottom-up traceability of the requirements from the SUT method. In the center of Figure 4.2 lies the representation of the method *“person_image_link”* presented earlier in Table 4.2. From this visualization it is possible to identify the requirements impacted by the *“person_image_link”* method. This information allows the testing team to know the magnitude of the impact of a particular change in the method’s source code.

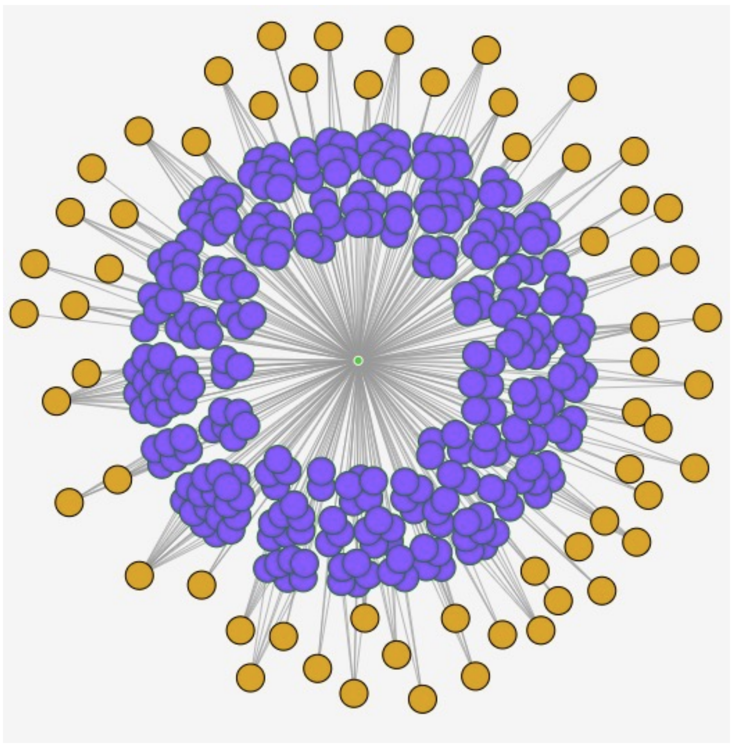


Figure 4.2: Traceability perspective of method *"person_image_link"*.

This set of views and information from software entities enables the application of various test case prioritization techniques, from black-box techniques to white-box techniques. The works by [12] and [15] discuss some of the common test case prioritization techniques. It was observed that the data obtained from our Feature-Trace approach allow the application of several of the mentioned techniques. Among white-box prioritization techniques, the Feature-Trace approach supports both *Total Coverage* based techniques, where test cases covering the largest number of units of source code are prioritized, as well as techniques based on *Additional Coverage*, which seeks to select test cases that cover the largest number of entities not yet covered by the set of test cases already executed. The approach also enables the use of prioritization techniques that are based on the diversity of the generated trace, using, for example, Global Maximum Distances algorithms [49]. About black-box test case prioritization techniques, the Feature-Trace approach makes it possible to use techniques based on requirements [50] and using concepts such as OP [1].

4.2 Evaluation Step 2

The main goal of the second validation step is to analyze the behavior and feedback of developers who have had access to the approach and tool proposed in this paper. It is necessary as the first validation step ensures Feature-Trace's ability to extract the required information and present it to all project stakeholders. However, only with the first validation step, we cannot guarantee the usefulness of Feature-Trace, that is, in practice, what are the advantages of using the proposed approach? In this sense, we chose to apply the proposed approach in real development environments, extracting the feedback from those involved.

At this stage, 18 students from the Computer Science course at the University of Brasilia were selected to use the tool and approach proposed, in order to identify the perception of those involved in using our approach. The students involved were studying around the third to fourth year of the Computer Science course and used the approach proposed in a subject project where the goal was evolve a software product to promote a real application context, with customers and needs well defined.

The evaluation process was defined in a way that sought to extract some information from the participants before using the proposed approach, so the participants had access to the tool and the approach presented. After using the tool, participants were asked to answer a new questionnaire so that it could be used as a comparison object between using or not using the proposed approach. Below are the questions presented in the first questionnaire (before using the proposed approach):

- **Question 1:** *Try to list, in order of priority, the methods that should be tested first in your project.*

  The main purpose of this question is to verify if the participants really know the software product they are working on. Someone who really knows the product and its importance to the user could list the elements of source code that deserve priority in comparison to other methods. In this case, the participant would point out, for example, if there are too many tests for a particular method, lack of tests in other methods, among other characteristics.

- **Question 2:** *What was the main criterion used for ordering the methods in the previous question?*

  This question seeks to understand what were the main criteria used to define the prioritization of test cases. We sought to compare this response to the response obtained after using the *Feature-Trace* tool by analyzing the differences.

- **Question 3:** *How would you rate the quality of the test suite present in your project?*

  It aims to analyze the participant's view of the current set of test cases in their project. A developer who knows the software product thoroughly can evaluate the quality of the present test case set. From this answer, the user will have access to the data presented by Feature-Trace and will be able to reassess the current set of test cases.

- **Question 4:** *Explain how you arrived at the answer to the previous question.*

  This question seeks to identify which criteria are used by developers when evaluating a set of test cases. What characteristics make a set of test cases appropriate or not?

- **Question 5:** *Can you list the methods that have been most tested so far? If so, how?*

  This question aims to create a list that will be used to make a comparison with the answers made after using the *Feature-Trace* tool and approach.

After obtaining this information, the participants used the proposed tool and approach to study and know their software. Questions 1 through 5 remain the same, they aim to analyze the difference between the developer's views before and after using the proposed approach. After question 5, the objective becomes feedback on the usefulness of the approach presented. Following are the applied questions:

- **Question 1:** *Try to list, in order of priority, the methods that should be tested first in your project.*

  Seeks to analyze differences from written responses before learning about the *Feature-Trace* approach and tool.

- **Question 2:** *What was the main criterion used for ordering the methods in the previous question?*

  Seeks to analyze differences from written responses before learning about the *Feature-Trace* approach and tool.

- **Question 3:** *How would you rate the quality of the test suite present in your project?*

  Seeks to analyze differences from written responses before learning about the *Feature-Trace* approach and tool.

- **Question 4:** *Explain how you arrived at the answer to the previous question.*

  Seeks to analyze differences from written responses before learning about the *Feature-Trace* approach and tool.

- **Question 5:** *Can you list the methods that have been most tested so far? If so, how?*

  Seeks to analyze differences from written responses before learning about the *Feature-Trace* approach and tool.

- **Question 6:** *Does feature-trace make the test case prioritization and selection process easier?*

It seeks to extract the participant's view of the support provided by the *Feature-Trace* tool regarding the prioritization and selection of test cases.

- **Question 7:** *Does the data presented represent well the test situation of the analyzed project?*

  The answers for this question get the participant's view on the representation of the existing test case set in the related project. Thus, it is analyzed if the *Feature-Trace* tool is able to represent well the current situation of the project, referring to the set of test cases. From this, we seek to know if the proposed tool allows the distribution of knowledge about the test case set to all participants involved in the software project.

- **Question 8:** *Can the data presented contribute to the maintenance and evolution activities of the analyzed project?*

  This question analyzes the user's view of the contribution of the proposed approach and tool in the software maintenance and evolution process.

- **Question 9:** *Can the data presented help to analyze the quality of the test case set of the analyzed project?*

  From this question it will be possible to draw the participant's view of the support provided by the *Feature-Trace* tool when referring to the quality assessment of the existing test case set.

- **Question 10:** *Would you have any suggestions for improving the Feature-Trace approach and tool?*

  This question seeks to identify points of improvement in the approach and tool proposed in this paper.

4.2.1 Data Analysis

The starting point of the data analysis refers to Question 3 raised, where it was sought to identify the view of the developers concerning the quality of the set of test cases in their project. It was observed that 11.1% of the developers rated their set of test cases with a maximum score, that was equivalent to number 5, 44.4% rated the tests with a score of 4, 33.3% rated it with a score of 3, and 5.6% gave a minimum score to set of test cases, that was equivalent to number 1.

After extracting this and other information, the participants were submitted to the use of the Feature-Trace tool to analyze the data analysis provided by the tool. We observed that the evaluation carried out after using the approach proposed in this work generated an increase of 50% in the number of grades 5 (maximum) in the quality of the test cases. Likewise, we observed a rise of grades 1 (minimum) in the quality of the set of test cases, increasing 100%. Grade 4 had a decrease of 25%, grade 3 had a decrease of 14.2% and grade 2 had a decrease of 100%, as show in Figure 4.3. That Figure objectively presents the differences between the evaluations carried out before and after using the *Feature-Trace* tool.

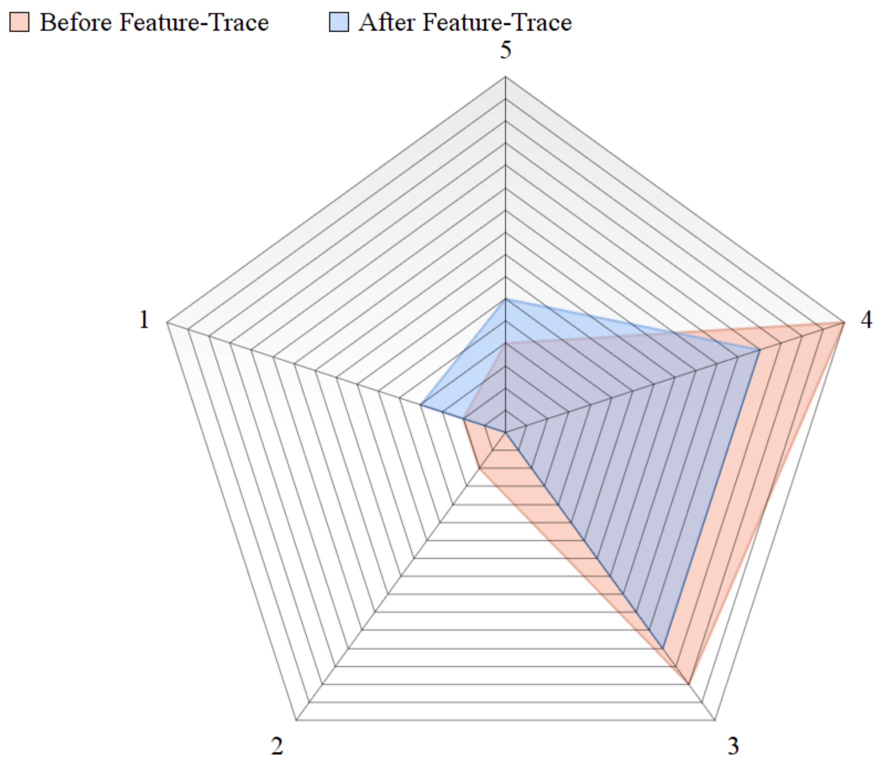


Figure 4.3: Participant perception of the quality of the existing test case set (Before and After using *Feature-Trace*).

When referring to the prioritization and selection of test cases, several criteria can be used to prioritize the next test cases to be created and/or executed. Before using the *Feature-Trace* approach, the criteria raised by the participants were: "Importance to the user", with 82.6%, "Code coverage", with 8.7%, and "Method complexity" and "Application security and Integrity" distributed in the remaining 8.7 %.

After using the *Feature-Trace* approach, developers were able to analyze new criteria and discuss the impact of each one on the quality of the generated software product. According to the data, this time, many other criteria not previously used were listed, showing an evolution in the degree of analysis performed by the participants when prioritizing/selecting test cases.

The criteria raised were: "Importance to the user", with 38.9%, "Code coverage", with 27.8% and six other criteria distributed equally among the remaining 33.3%. The new criteria raised are:

- Complex methods without sufficient testing;
- Probability of method occurrence and amount of existing tests
- Analysis of how the method is tested and its importance to the feature
- Probability of execution
- Requirements impacted by each method

The increase in the number of criteria used by the participants shows an exciting learning experience on the part of the developers. Criteria that made no difference in the process of prioritizing/selecting test cases came to be seen as important points that deserve attention.

Just as new criteria started to be analyzed by the participants for prioritization/selection of test cases, it is acceptable to state that these criteria also started to be used in the item "quality of the test case set" (Question 3). Thus, as shown in Figure 4.3, some changes were noticed in the participants' assessment of their set of test cases.

In relation to the participants' feedback on the approach presented in this work, Figure 4.4 presents the utility observed by the participants in three main criteria, that are: 1) usefulness in the prioritization process and Selection of test cases, 2) useful when referring to the analysis of the set of existing test cases, and 3) the utility during maintenance activities and software evolution. Each of the three items (Questions 6, 7, and 8) was obtained from the registration of a note on a scale of 1 to 5, where one represents "no utility", and 5 represents "maximum utility".

It can be seen from Figure 4.4 that the three criteria analyzed are distributed between values 3 and 5, being mostly concentrated on "maximum utility". More specifically, in the item Prioritization and Selection of test cases, 77.8% of the participants rated the utility of the approach/tool with a score of 5 (maximum), followed by 16.7% with a rating of 4 and 5.6% defining a score of 3 for utility of *Feature-Trace*. In terms of the ability of the proposed tool to extract and present information about the current situation of the set of test cases present in the software product, 83.3% of the participants rated this capacity with a score of 5, 11.1% of the participants rated it with a score of 4 and 5.6% rated it with a score of 3.

Concerning the maintenance and evolution of software, 83.3% of the participants rated their usefulness in maintenance and evolution with grade 5, followed by 11.1% with grade 4 and 5.6% with grade 3. None of the participants involved in the research registered a grade 1 or 2 in these items, which shows the usefulness of the Feature-Trace approach and the development of the support tool.

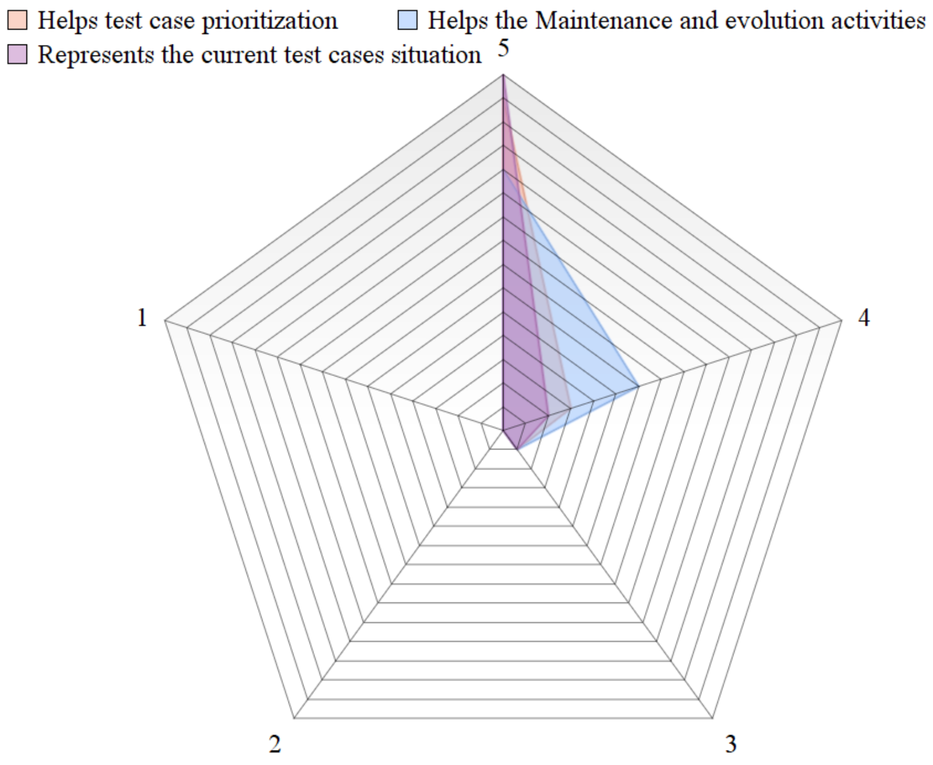


Figure 4.4: Feedback about Feature-Trace usefulness.

4.3 Discussion

Throughout the work, several discussions were raised related to the process of defining the order of prioritization of test cases, as well as the importance of knowledge of source code. Taking these two main points into consideration, a proposed approach, along with a support tool, was presented to enable a test case prioritization process based on knowledge of the source code, existing test case set, and requirements involved in the project. With this information available, we enable an environment where everyone involved in the development process can talk and jointly decide how to prioritize and select test cases, taking into account the criteria set in the context of the project.

The work evaluation was divided in two stages: in the first stage, we sought to evaluate the ability of the Feature-Trace tool to extract relevant information for the adoption of the proposed approach. At this stage, the “Diaspora” system was used as a case study. We chose to use “Diaspora” because it is a widely recognized open-source system in the community and is based on the use of the BDD approach as a way to record and validate system requirements (which is a prerequisite for adopting the proposed approach). The second validation step sought to analyze the feedback given by developers who had access to the approach and tool generated in this research.

Regarding the capability of the Feature-Trace tool, we observe from the results presented that the tool can extract several useful information for the proposed approach, among them:

- **Method Spectrum**

  Number of test cases exercising a given method. With this information, the team can easily understand which methods have been tested the most and which still deserve more attention for lack of testing.

- **Test Case Method Coverage**

  Represents the number of methods executed by a given test case. With this information, we can identify the degree of specificity of each test case. Some test cases prove to be more specific by analyzing only one or a few methods. In contrast, other test cases may even check hundreds of methods in one run, behaving like more generic test cases, or closer to a black-box pattern of a test case.

- **Feature Spectrum**

  Represents the number of test cases that seek to verify a feature. With this information, the team can discuss and decide the distribution of test cases by requirement. This debate can take into consideration the degree of importance that the business area gives to each system requirement.

- **Impacted Features**

  This information presents to the developer the number of features that may be impacted by a change in a particular software method. In addition to the number of impacted features, the user can analyze which features are affected by each method. This information is considered highly relevant as developers, and all people involved in the project can easily understand the threats present in a specific code change and are very useful in software maintenance and evolution activities.

- **Cyclomatic Complexity , ABC Score and Number of lines of each method**

These three metrics are used to analyze the degree of complexity of a given method. In this context, this information is very useful due to the fact that participants can take into account the effort required to implement test cases in certain methods. Thus, with this information it is possible to deduce, even minimally, the effort spent to obtain the benefit presented by the union of the other metrics extracted by the Feature-Trace tool. In this way, the project team is able to think about the cost-benefit concept when defining a prioritization order and selection of test cases.

- **Operational Profile**

  Obtaining the Operational Profile of the software, as described in Chapter 3, is done in a semi-automated manner, based on the structure of the requirements defined from the Behavior Driven Development (BDD) approach. With this information in hand, we can sort all code entities by taking into account the likelihood of each entity occurring during the execution of software in the production environment. That is, knowing all probabilities, seeking acceptable reliability becomes a more practical and straightforward activity since software reliability is given by the number of failures over a given period.

  Operational Profile information alone would be beneficial for a software quality assurance process. However, as highlighted by [17], the pure and straightforward use of the Operational Profile can stabilize the reliability growth after many test cycles, given that following only this strategy, only the methods most likely to occur are being prioritized, forgetting those with a low probability of occurrence. Although these inferior probability methods do not have much impact on software reliability, at some point, the growth of software reliability will depend on these methods as all high probability methods have already been thoroughly tested.

From the verification of the ability of the tool to extract the information mentioned above, we seek to analyze projects that would be possible to present the data generated to those involved in the project and get feedback from each participant about the use of the tool. Thus, the idea of performing the second stage of validation of the approach and tool proposed in this research arises.

Four academic projects made in Ruby on Rails by students of the sixth semester of Computer Science from the University of Brasilia (UnB) were used. In total, eighteen students were interviewed. The interview was conducted in two parts, the first to understand how these students analyze a software project and prioritize test cases at each test cycle (questions 1-5). After conducting the first part of the interview, the students were submitted to the Feature-Trace approach and tool, analyzing the data generated from their projects. After data analysis, the second part of the interview was conducted, which sought to repeat the questions made in part 1, in order to compare the students' views regarding the quality, prioritization, and selection of test cases before and after the use of the approach proposed in this research. The second interview stage also included some questions aimed at extracting participants' feedback regarding the usefulness of the tool and approach proposed in this paper.

We could see a difference of the criteria used for prioritizing test cases before and after using Feature-Trace. We observed from the results presented that students' view before using the Feature-Trace approach was limited to only four criteria: *Code Coverage, Method Complexity, Importance to the User and Application Security and Integrity.* However, after knowing the approach and the Feature-Trace tool, the criteria used by the students showed itself much more specific: *Code Coverage, Method Complexity, Importance to the User, Complex methods without sufficient testing, Probability of method occurrence and amount of existing tests, Analysis of how the method is tested and its importance to the feature, Probability of execution, and Requirements Impacted by each method.* This result highlights one of the advantages of using the proposed approach: *Presenting stakeholders with information that can contribute to the prioritization and selection of test cases, enriching the discussion among the team.*

As we can see from the data presented by Figure 4.4, the feedback regarding the utility of the tool presented itself quite positive, given that no grade 1 or 2 was given to any criterion of utility of the tool, having, most of them, 5 and 4. This result shows the degree of usefulness of the tool and approach proposed in this work, highlighting well the contributions generated by the research in question. Knowing the software product as a whole is a matter of paramount importance for carrying out software development, maintenance, and testing activities. The *Feature-Trace* approach has as main objective to present relevant information so that those involved in the project can know the highest amount of information about the product in question.

4.4 Threats to Validity

The results presented are subject to some threats to the validity of the case study and should be highlighted.

Related to the *construct validity*, the generation of OP, in the first validation step, is based on the hypothetical usage profiles, which make unrealistic the comparative analysis between the degree of importance of the source code unit and the number of test cases exercised. However, since the objective of the study is to present the capability to support the prioritization of test cases of the SUT, these hypothetical data do not affect the validity of the obtained results and the performed analysis. If the OP has real data, the analysis becomes realistic and the prioritization process can rely on the actual data of the involved OPs.

Regarding the *internal validity*, we highlight the fact that we used only some of the metrics useful for the process of prioritizing test cases. However, we sought to select at least one metric related to the impact on reliability (OP), three metrics related to the testing effort (complexity) and one metric referring to the current coverage of each computational unit. We understand that with this set of metrics, it is already possible to make a process of prioritization and selection of test cases feasible, with essential discussions among team members based on these data.

Concerning *external validity*, we considered as a threat the first validation step is focused only on Diaspora. Further case studies must be performed so that the ability to generalize our results can be validated. However, we should note that Diaspora is a software used by real software engineering development teams and has been focus of significant number of contributions in the open source software community. Another threat refers to the second validation step, given that only projects carried out within the academy were used as a basis for extracting feedback from participants. In the next steps of the research, we will seek to carry out this analysis in a market environment, also enabling a comparison between market and academic feedbacks.

Finally, to make feasible the replication of our outcomes in the case study, all the artifacts of the evaluation process are publicly available on Github[7].

[7]https://github.com/FeatureTrace/SBES2019

Chapter 5

Related Work

Several contributions in the literature have proposed means to enable the obtaining of the OP, while minimizing the effort for its application. A widely used strategy is the analysis of system usage logs from the production environment [51, 52, 53]. In order for these strategies to be applied, the SUT must be used by its actual users, unlike the Feature-Trace approach, which makes it possible to obtain the OP from the extraction of information from the BDD artifacts.

Among those contributions that aim to approximate the software requirements to the test artifacts, we highlight the work proposed by Per Runeson et al. [54], which suggests the relationship between the Use-Case Model and OP, minimizing the effort required to create the two artifacts. In your work, [54] presents two approaches for integration between the Use-Case Model and the *Operational Profile*: *transformation* and *extension* of the Use-Case Model. The *transformation* approach receives the Use-Case Model as *input*, returning a second model containing information from the *Operational Profile* beside of Use-Case Model information. The *extension* approach uses the Use-Case Model itself to include information from *profile*, integrating the two models into just one. According In the study, the two approaches generate clear benefits concerning the effort spent on modeling, developing, and testing the software. Meantime, both approaches have their advantages and disadvantages, the main advantage of *extension* being minimizing effort of modeling, since, in this approach, only one model is generated as a result, including both information. Regarding the approach of *transformation*, the main advantage is about the conflict of responsibilities, since the Use-Case Model is concerned with the important features for the user and the *Operational Profile* is concerned with the entities most likely to be executed. Thus, it is interesting to keep this information in different artifacts, facilitating the distinction of criteria such as **importance to the user** and **probability of execution** [54].

There are also other contributions which map further requirements artifacts alongside the concept of OP [55, 56]. Unlike our Feature-Trace approach, the execution of those requirements artifacts based on Use-Case Models is performed manually, as opposed to using the BDD approach. That is, in the Feature-Trace approach, the generation and monitoring of the generated data can be done in an automated way. Moreover, BDD is also a living documentation, while use case requires the constant update from the requirements team.

In addition, some contributions focus on the process of prioritizing test cases. Bertolino et al. [17] propose the *covrel* approach, where the concepts of OP (black-box information) and Program Spectrum (white-box information) are used to maximize the growth of SUT reliability throughout the testing process. According [17], the approach ponders the information of the OP with information about the execution path of the SUT, maximizing the attention to entities with a low probability of occurrence and that have been little exercised by the set of existing test cases. In the sequel, Miranda et al.[57] evaluate the use of operational coverage (OP weighted with Program Spectrum) as a criterion of adequacy and selection of test cases in an OP-based test context. In their study, it is confirmed that the use of operational coverage proved to be more efficient in relation to the size of the test suite by the number of defects identified compared to the traditional OP.

About the variety of techniques and strategies of prioritization of test cases that can be used during a process of prioritization or selection of test cases, Ouriques et al. [58] make a survey of some techniques related to the context of model-based testing (MTB). According to the presented result, *"there is no clear best performer among them"*. It was observed that the performance of each technique depends on the context involved, that is, the choice of technique and strategy of prioritization of test cases should be based on the characteristics included in the project. In this sense, the Feature-Trace approach allows the selection of the technique/strategy desired by the test team, providing a set of information relevant to the application of prioritization and selection techniques.

The use of testing tools made available as Web Services was discussed by [59] when proposing the tool JaBUTiService, presenting some benefits related to this context: Ease of use, Availability, Version Control, Integration, Orchestration, and Choreography of software engineering services and Comparison. Due to the architectural character of the tool proposed in this work, we have added some of these advantages, which deserve future evaluation. The JaBUTiService tool tries to centralize in a server some activities related to the structural test, allowing the integration of different systems and minimizing the effort related to the test environment. In comparison to this strategy, our approach supports analysis of test cases based on criteria beyond code coverage, including software OP, unlike JaBUTiService.

The main contribution of the current research is related to obtaining and using the

Operational Profile of the software in a semi-automated way. In this sense, Table 5.1 presents the relationship between the main works focused on the generation and/or use of the Operational Profile to prioritize test cases.

Table 5.1: Related Works.

Work	**Approach**	**Phase**	**Requirements Traceability**	**Getting OP**
Nagappan et al. [51]	Operational Profile	Released	No	Log analysis
Hassan et al. [52]	Operational Profile	Released	No	Log analysis
Yamany et al. [53]	Operational Profile	Released	No	Usage monitoring with multi-agents.
Runeson et al. [54]	Operational Profile + Use Case Model	Requirements Elicitation	Yes	Extension of Use Case model with OP
Bertolino et al. [17]	Operational Profile + Program Spectrum	All life Cycle	No	Does not support.
Feature Trace	**Operational Profile + Program Spectrum + BDD**	**All life cycle**	**Yes**	**Tagged BDD scenarios**

Chapter 6

Conclusion and Future Work

The approach presented in this paper, Feature-Trace, is based on the software traceability, derived from the use of the BDD approach, to extract relevant information that allows the application of several strategies and techniques of prioritization and selection of test cases, including the concept of OP. According to the presented case study, using the Diaspora social network, the proposed approach allows the generation of the OP with low effort and naturally integrated into the development of software. Through the case study, it was possible to observe that, since the approach aims to automate a large part of the analysis and generation of the OP, the team involved can aggregate, without much effort, relevant information to guide the process of software maintenance, evolution, and testing. In this way, the Feature-Trace approach extracts and presents some information and analyzes so that the team decides how its test case prioritization/selection strategy will be guided.

It was possible to observe from the evaluation with 18 participants that the approach presented and the support tool implemented bring significant benefits to the process of prioritizing/selecting test cases, to the activities of evaluating the quality of test cases and during Maintenance activities and software evolution. We obtained positive feedback, evidencing some of the contributions of this research.

As future work, we plan to evaluate the Feature-Trace approach in a market environment, given that the evaluation carried out in this research was primarily centered on the academic environment. Also, we are working on the generation of new modules that use the data generated by the *Feature-Trace* tool to carry out other activities related to the software development process. We highlight the activities of Test Case Generation in an automated way and Automated Selection of Test Cases based on modifications of the source code. Also, we wish to evaluate the approach about the benefits obtained with the use of the proposed tool in relation to its architectural characteristics.

Reference List

[1] Musa, John D: *Operational profiles in software-reliability engineering.* IEEE software, (2):14–32, 1993. vii, 2, 11, 12, 13, 14, 25, 27, 31, 41

[2] Cucumber: *Gherkin reference.* `https://docs.cucumber.io/gherkin/reference/#keywords`, visited on 2018-09-09. vii, 17

[3] Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli: *Fundamentals of software engineering.* Prentice Hall PTR, 2002. 1

[4] Jiménez, Miguel, Mario Piattini, and Aurora Vizcaíno: *Challenges and improvements in distributed software development: A systematic review.* Advances in Software Engineering, 2009:3, 2009. 1

[5] Catal, Cagatay and Banu Diri: *A systematic review of software fault prediction studies.* Expert systems with applications, 36(4):7346–7354, 2009. 1

[6] Gómez, Oswaldo, Hanna Oktaba, Mario Piattini, and Félix García: *A systematic review measurement in software engineering: State-of-the-art in measures.* In *International Conference on Software and Data Technologies*, pages 165–176. Springer, 2006. 1

[7] Delamaro, Marcio, Mario Jino, and Jose Maldonado: *Introdução ao teste de software.* Elsevier Brasil, 2017. 1

[8] Myers, Glenford J, Corey Sandler, and Tom Badgett: *The art of software testing.* John Wiley & Sons, New Jersey, USA, 2011. 1, 2

[9] Rempel, Patrick and Parick Mäder: *Preventing defects: The impact of requirements traceability completeness on software quality.* IEEE Transactions on Software Engineering, 43(8):777–797, 2017. 2, 3, 5

[10] Adrion, W Richards, Martha A Branstad, and John C Cherniavsky: *Validation, verification, and testing of computer software.* ACM Computing Surveys (CSUR), 14(2):159–192, 1982. 2

[11] Catal, Cagatay and Deepti Mishra: *Test case prioritization: a systematic mapping study.* Software Quality Journal, 21(3):445–478, 2013. 2, 10, 11

[12] Henard, Christopher, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon: *Comparing white-box and black-box test prioritization.* In *Software Engineering*

(ICSE), 2016 IEEE/ACM 38th International Conference on, pages 523–534. IEEE, 2016. 2, 30, 41

[13] Ebert, Christof and James Cain: *Cyclomatic complexity.* IEEE software, 33(6):27–29, 2016. 2

[14] Noor, Tanzeem Bin and Hadi Hemmati: *A similarity-based approach for test case prioritization using historical failure data.* In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68. IEEE, 2015. 2

[15] Miranda, Breno, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino: *Fast approaches to scalable similarity-based test case prioritization.* In *Proceedings of the 40th International Conference on Software Engineering*, pages 222–232. ACM, 2018. 2, 30, 41

[16] Musa, John D: *The operational profile.* In *Reliability and Maintenance of Complex Systems*, pages 333–344. Springer, 1996. 2, 10, 11

[17] Bertolino, Antonia, Breno Miranda, Roberto Pietrantuono, and Stefano Russo: *Adaptive coverage and operational profile-based testing for reliability improvement.* In *Proceedings of the 39th International Conference on Software Engineering*, pages 541–551. IEEE Press, 2017. 3, 4, 9, 21, 26, 34, 49, 53, 54

[18] Bjarnason, Elizabeth, Per Runeson, Markus Borg, Michael Unterkalmsteiner, Emelie Engström, Björn Regnell, Giedre Sabaliauskaite, Annabella Loconsole, Tony Gorschek, and Robert Feldt: *Challenges and practices in aligning requirements with verification and validation: a case study of six companies.* Empirical Software Engineering, 19(6):1809–1855, 2014. 3, 5, 8

[19] Eisenbarth, Thomas, Rainer Koschke, and Daniel Simon: *Locating features in source code.* IEEE Transactions on software engineering, 29(3):210–224, 2003. 3

[20] Liu, Dapeng, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich: *Feature location via information retrieval based filtering of a single scenario execution trace.* In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007. 3

[21] Eaddy, Marc, Alfred V Aho, Giuliano Antoniol, and Yann Gaël Guéhéneuc: *Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis.* In *2008 16th IEEE International Conference on Program Comprehension*, pages 53–62. Ieee, 2008. 3

[22] Beck, Fabian, Bogdan Dit, Jaleo Velasco-Madden, Daniel Weiskopf, and Denys Poshyvanyk: *Rethinking user interfaces for feature location.* In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 151–162. IEEE Press, 2015. 3

[23] Li, Yi, Chenguang Zhu, Julia Rubin, and Marsha Chechik: *Fhistorian: Locating features in version histories.* In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 49–58. ACM, 2017. 3

[24] Inayat, Irum, Siti Salwah Salim, Sabrina Marczak, Maya Daneva, and Shahaboddin Shamshirband: *A systematic literature review on agile requirements engineering practices and challenges.* Computers in human behavior, 51:915–929, 2015. 3

[25] North, Dan: *Introducing bdd*, 2003. `https://dannorth.net/introducing-bdd/`, visited on 2018-06-15. 3, 25

[26] Pereira, Lauriane, Helen Sharp, Cleidson de Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos: *Behavior-driven development benefits and challenges: reports from an industrial study.* In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, page 42. ACM, 2018. 4

[27] Fazzolino, Rafael and Genaína Nunes Rodrigues: *Feature-trace: Generating operational profile and supporting testing prioritization from bdd features.* In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 332–336, 2019. 5

[28] Sommerville, Ian *et al.*: *Software engineering.* Addison-wesley, 2007. 7

[29] Fox, Armando, David A Patterson, and Samuel Joseph: *Engineering software as a service: an agile approach using cloud computing.* Strawberry Canyon LLC India, 2014. 7, 8

[30] North, Dan *et al.*: *Introducing bdd.* Better Software, March, 2006. 8

[31] Whalen, Michael W, Ajitha Rajan, Mats PE Heimdahl, and Steven P Miller: *Coverage metrics for requirements-based testing.* In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36. ACM, 2006. 9

[32] Smart, John Ferguson: *BDD in Action.* Manning Publications, 2014. 9, 19

[33] Singh, Leena and Leonard Drucker: *Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout.* Springer Science & Business Media, 2007. 9

[34] Jorgensen, Paul C: *Software testing: a craftsman's approach.* Auerbach Publications, 2013. 9

[35] Zhu, Hong, Patrick AV Hall, and John HR May: *Software unit test coverage and adequacy.* Acm computing surveys (csur), 29(4):366–427, 1997. 9

[36] Miranda, Breno and Antonia Bertolino: *Does code coverage provide a good stopping rule for operational profile based testing?* In *Automation of Software Test (AST), 2016 IEEE/ACM 11th International Workshop in*, pages 22–28. IEEE, 2016. 9, 10

[37] Reps, Thomas, Thomas Ball, Manuvir Das, and James Larus: *The use of program profiling for software maintenance with applications to the year 2000 problem.* In *Software Engineering—Esec/Fse'97*, pages 432–449. Springer, 1997. 10, 19

[38] Yoo, Shin and Mark Harman: *Regression testing minimization, selection and prioritization: a survey.* Software Testing, Verification and Reliability, 22(2):67–120, 2012. 10

[39] Beck, Kent: *Test-driven development: by example.* Addison-Wesley Professional, 2003. 14

[40] Evans, Eric: *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004. 14

[41] Smart, John Ferguson: *BDD in Action: Behavior-driven development for the whole software lifecycle.* Manning, 2015. 16

[42] Solis, Carlos and Xiaofeng Wang: *A study of the characteristics of behaviour driven development.* In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387. IEEE, 2011. 16

[43] Oliveira, Gabriel, Sabrina Marczak, and Cassiano Moralles: *How to evaluate bdd scenarios' quality?* In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 481–490, 2019. 19

[44] Harrold, Mary Jean, Gregg Rothermel, Rui Wu, and Liu Yi: *An empirical investigation of program spectra.* In *Acm Sigplan Notices*, volume 33, pages 83–90. ACM, 1998. 19

[45] Fitzpatrick, Jerry: *Applying the abc metric to c, c++, and java.* More C++ gems, pages 245–264, 1997. 26

[46] McCabe, Thomas J: *A complexity measure.* IEEE Transactions on software Engineering, (4):308–320, 1976. 26

[47] Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk: *Feature location in source code: a taxonomy and survey.* Journal of software: Evolution and Process, 25(1):53–95, 2013. 30

[48] Zhang, Lingming, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei: *Bridging the gap between the total and additional test-case prioritization strategies.* In *Proceedings of the 2013 International Conference on Software Engineering*, pages 192–201. IEEE Press, 2013. 31

[49] Henard, Christopher, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon: *Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines.* IEEE Transactions on Software Engineering, 40(7):650–670, 2014. 31, 41

[50] Arafeen, Md Junaid and Hyunsook Do: *Test case prioritization using requirements-based clustering.* In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 312–321. IEEE, IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013. 31, 41

[51] Nagappan, Meiyappan, Kesheng Wu, and Mladen A Vouk: *Efficiently extracting operational profiles from execution logs using suffix arrays.* In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 41–50. IEEE, 2009. 52, 54

[52] Hassan, Ahmed E, Daryl J Martin, Parminder Flora, Paul Mansfield, and Dave Dietz: *An industrial case study of customizing operational profiles using log compression.* In *Proceedings of the 30th international conference on Software engineering*, pages 713–723. ACM, 2008. 52, 54

[53] Yamany, Hany EL and Miriam AM Capretz: *A multi-agent framework for building an automatic operational profile.* In *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, pages 161–166. Springer, 2007. 52, 54

[54] Runeson, Per and Björn Regnell: *Derivation of an integrated operational profile and use case model.* In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 70–79. IEEE, 1998. 52, 54

[55] Simmons, Erik: *The usage model: Describing product usage during design and development.* IEEE software, 23(3):34–41, 2006. 53

[56] Schuur, Henk van der, Slinger Jansen, and Sjaak Brinkkemper: *A reference framework for utilization of software operation knowledge.* In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 245–254. IEEE, 2010. 53

[57] Miranda, Breno and Antonia Bertolino: *An assessment of operational coverage as both an adequacy and a selection criterion for operational profile based testing.* Software Quality Journal, 26(4):1571–1594, 2018. 53

[58] Ouriques, Joao Felipe S, Emanuela G Cartaxo, and Patrícia DL Machado: *Test case prioritization techniques for model-based testing: a replicated study.* Software Quality Journal, pages 1–32, 2018. 53

[59] Eler, Marcelo Medeiros, Andre Takeshi Endo, Paulo Cesar Masiero, Marcio Eduardo Delamaro, Jose Carlos Maldonado, Auri Marcelo Rizzo Vincenzi, Marcos Lordello Chaim, and Delano Medeiros Beder: *Jabutiservice: a web service for structural testing of java programs.* In *2009 33rd Annual IEEE Software Engineering Workshop*, pages 69–76. IEEE, 2009. 53