UCLouvain

École polytechnique de Louvain

Testing the design of context-oriented software through mutation testing

Author: **Audric DECKERS**
Supervisors: **Kim MENS, Pierre MARTOU**
Reader: **Nicolás CARDOZO**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

In recent years, the Context-Oriented Programming paradigm has emerged, driven by the growing interest in context-aware systems. These systems can adapt their behaviour based on contextual information gathered from the environment at runtime. The RELEASeD research group in UCLouvain has developed a software development approach that combines context-oriented programming with feature modelling to represent behavioural variations and the contextual environment in two separate models. However, designing these models can be challenging due to the inherent complexity of context-oriented systems. This master thesis introduces a mutation-based recommendation system that challenges the design choices of the developer and ensures the accurate representation of their envisioned system. The system efficiently generates a manageable set of questions in a short time, with a linear complexity in the number of contexts and features in the model.

Acknowledgments

I am extremely grateful to my two supervisors, Prof. Kim Mens and Ph.D. student Pierre Martou for their availability, precious advice, and constructive comments throughout this year and until the end. Our weekly meetings were always relevant and always with constructive feedback.

I would also like to extend my thanks to my family and friends for their unwavering support, love, and encouragement throughout my academic journey. Your support has been a source of motivation and inspiration for me.

On a fun note, I would like to express my gratitude to OpenAI for its exceptional generative tools, and to DeepL for helping me write this master thesis.

Contents

1 Introduction **5**

2 Background **7**
2.1 Feature Modelling 7
2.1.1 Feature 7
2.1.2 Feature model 8
2.1.3 Converting Feature models to propositional logic 9
2.2 Feature-based context-oriented programming 10
2.2.1 Context-Oriented Programming 11
2.2.2 Feature-based context-oriented programming approach . . . 11
2.2.3 Context-Feature-Mapping model 13
2.2.4 Converting CFM model to conjunctive normal form 14
2.3 SAT solver 15
2.4 Mutation Testing 16
2.4.1 Traditional mutation testing 16
2.4.2 Feature-based mutation testing 17
2.5 Background Summary 20

3 Running example **21**
3.1 Smart messaging application 21
3.2 Lexicon 23
3.2.1 Context model 23
3.2.2 Feature model 23
3.2.3 Mapping model 24

4 Motivations, Problem Statement and Analysis **25**
4.1 The importance of software testing 25
4.2 The complexity within FBCOP systems 26
4.3 The choice of selective mutation testing 27
4.4 Overview of the Problem Statement 28

5 Theoretical Foundations and Approach **29**
5.1 Formatting the CFM model 29
5.2 Designing FBCOP-Specific mutant operators 31
5.2.1 Adapting existing mutant operators to CFM models 31
5.2.2 Deriving mapping-based mutant operators 33
5.3 Generating useful recommendations 34
5.3.1 Preventing the generation of equivalent mutants 35
5.3.2 Introducing constraint hierarchy in CFM models 36
5.3.3 Prioritizing mutants for presentation 37
5.4 Mapping mutant operators to questions 37
5.5 Overview of the different concepts 38

6 Recommendation system **39**
6.1 Recommendation system architecture 39
6.1.1 Data acquisition and processing module 40
6.1.2 Connected pairs generation module 41
6.1.3 Mutants generation module 42
6.1.4 Mutant Q&A validation module 42
6.2 Results of our Prototype 43
6.2.1 Raw results 43
6.2.2 Discussion and analysis 44

7 Validation **45**
7.1 Validation methodology 45
7.2 Raw Results 45
7.3 Discussion and analysis 46
7.3.1 Efficiency and rapidity 46
7.3.2 Ease of use 47
7.4 Threats to validity 47
7.5 Limitations 47

8 Future Work **48**
8.1 Centralized testing application for feature-based context-oriented programming systems 48
8.2 Pushing further our recommendation system 49
8.3 Investigate other mutation testing locations 49

9 Conclusion **50**

Chapter 1

Introduction

Context-Oriented Programming (COP) is a programming paradigm that enables the creation of context-aware applications. Such applications can adapt their behaviour based on contextual information gathered from the environment at runtime [23]. Over the past decade, several COP languages have been proposed to facilitate the development of such applications. Duhoux et al. created a unified software development approach called Feature-Based Context-Oriented Programming (FBCOP) [8, 10] that integrates elements of COP, Feature Modeling, and Dynamic Software Product Lines.

In the FBCOP approach, behavioural variations of a system are represented in a feature model, while the contextual environment in which the system operates is represented in a separate context model. These models are linked by a mapping model that specifies the relationships between contexts and features. These three components together form a Context-Feature-Mapping (CFM) model. Specifically, the activation or deactivation of a given context can trigger the inclusion or removal of certain features in the system, as well as the replacement or extension of other features. This allows the system to adapt its behaviour to the changing contextual environment at runtime.

The combination of the dynamic addition or removal of highly variable behavioural variations and the ability for these variations to refine each other causes a quite dynamic and intricate control flow that poses challenges for software testing. To address these challenges, dedicated testing approaches are required. Prior research in this field has already produced a visualisation tool to assist developers in managing the inherent complexity of such systems [9, 11, 12]. Additionally, research has been conducted on generating test suites using pairwise combinatorial interaction testing (CIT) to manage the exponential growth of configurations [21]. However, neither of these studies mainly focused on evaluating the design choices

made in CFM models to ensure that they accurately reflect the system envisioned by the designer. The correct representation of a FBCOP system is crucial to ensure that it meets the needs and requirements of the users. In the case of a messaging application, for example, a design error in the CFM model could result in an annoying situation, such as an intrusive ringing phone during a meeting, which should be avoided for the user's comfort. However, the consequences of a single design flaw in more critical applications such as a risk information system [9] or a medical application could be catastrophic. This highlights the crucial need for careful consideration during the design process of dynamically adaptive systems.

The focus of this master thesis is thus to examine the feasibility and relevance of using an alternative testing approach based on Mutation Testing (MT) to evaluate FBCOP systems. Our main contribution is the development of a recommendation system that aims to challenge the design choices made during the construction of CFM models. This system is based on selective mutation analysis [25] and the concept of distinguishing configurations (i.e. a configuration that is valid in an original model M but not in its mutant M', or vice versa) proposed by Arcaini et al. in 2015 [1, 2]. We are able to generate a reasonable number of questions in a short time, with a linear complexity in the number of contexts and features within the CFM model.

This master thesis is structured as follows: Chapter 2 provides an explanation of essential concepts such as features, feature modelling, Feature-Based Context-Oriented Programming (FBCOP), mutation testing, and SAT solvers. In Chapter 3, a smart messaging application is introduced as an illustrative example, which will be referenced throughout the master thesis as needed. Chapter 4 presents the motivations, the problem statement, and an analysis of the identified problem. Chapter 5 delves into the theoretical foundations and key concepts of our recommendation system. Chapter 6 reveals the architecture of our recommendation system, showcasing the results obtained from an initial version applied to our smart messaging application. Chapter 7 engages in a comprehensive discussion of the validation process for our system, including its effectiveness and limitations. Chapter 8 explores various avenues for future work, presenting potential directions for further development and improvement. Finally, Chapter 9 concludes the master thesis by summarizing the achievements, contributions, and findings.

Chapter 2

Background

In order to comprehend the key chapters of the master thesis, this section will provide the necessary background information. To properly assess the feasibility of mutation testing for testing FBCOP systems, it is essential to understand what mutation testing and FBCOP systems are, along with the key concepts needed to comprehend them. Thus, in section 2.1, we will define feature and feature models, followed by a short introduction of the FBCOP approach in section 2.2. In section 2.3 we will delve into the concept of SAT solvers, a concept on which a part of our approach is based. Then, in section 2.4, we will describe the mutation testing technique and ways to perform it on feature models. Lastly, in section 2.5, we will conclude by providing a concise summary of the key concepts and insights covered throughout this chapter.

2.1 Feature Modelling

2.1.1 Feature

We will begin by defining what a feature is, since it is a key element in representing the behaviour of FBCOP systems. A feature is defined by Kang et al. [19] as "*a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems.*". Taking for example the Car system represented in Figure 2.1, *Car body* and *Pulls trailer* are both features that respectively represent the body of a car and the ability of the *Car* to pull a trailer. *Car body* and *Pulls trailer* are thus characteristics of our *Car* system. In this master thesis, each feature will be represented as a rectangular node in a tree-like hierarchical structure, called a feature model. In these models, multiple features can be combined to create a complete system or product, such as a car. Each valid combination of features represents a specific configuration of the system or product (e.g. a particular car).

2.1.2 Feature model

The feature modelling notation is commonly used in software product lines to describe the commonalities and variabilities of a family of systems through a set of features and relationships between them. When an entire family of software systems is represented instead of a single system, it is called a Software Product Line (SPL). An example of feature model is given in Figure 2.1.

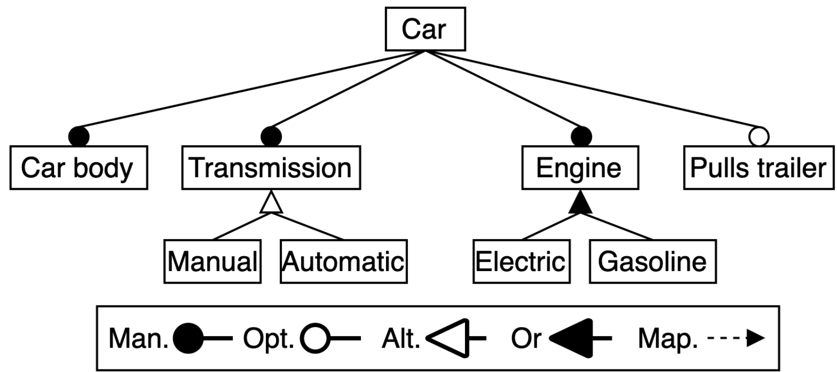


Figure 2.1: Basic Feature Model example representing a car system.

Considering the example of Figure 2.1, the *root* of the feature model is the *Car* itself, which has four child features: three **Mandatory** (*Car body, Transmission and Engine*) and one **Optional** (*Pulls trailer*). The *Transmission* feature has an **Alternative** constraint on its two child features, *Manual* and *Automatic*, and the *Engine* feature has an **Or** constraint on its two child features, *Electric* and *Gasoline*. Explanations and representations of possible parent-child relationships between features are described in Figure 2.2.

A configuration is considered valid if it satisfies all the constraints imposed by the model. For instance, in our car example, a valid configuration could include *Car*, *Car body*, *Transmission*, *Manual*, *Engine*, and *Gasoline*. Conversely, an invalid configuration would select both *Manual* and *Automatic* features, as it violates the alternative constraint which mandates that exactly one of the child features of *Transmission* should be selected.

Commonalities refer to essential features that are selected in any system configuration and are part of the core system. In the given example, mandatory features such as *Car body, Transmission*, and *Engine* are considered as *commonalities*. On the other hand, Variabilities are features that distinguish each software system as they may or may not be selected in a system configuration, and are not essential

for the proper functioning of the system. In the example, features such as *Pulls trailer*, *Manual*, *Automatic*, *Electric*, and *Gasoline* are categorized as *variabilities*.

FM Representation	FM Explanation	FM Representation	FM Explanation
Optional p c	When a parent feature p is selected, its child feature c may or may not be selected.	Or p c_1 ... c_n	When a parent feature p is selected, at least one of its child features (c_1... c_n) must be selected.
Mandatory p c	When a parent feature p is selected, its child feature c must also be selected.	Alternative p c_1 ... c_n	When a parent feature p is selected, exactly one of its child features (c_1... c_n) must be selected.
Note that in all cases, if a parent feature p is not selected, then none of its children can be selected either.			

Figure 2.2: Explanation and representation of simple parent-child relationships in a feature model.

2.1.3 Converting Feature models to propositional logic

This subsection will cover propositional formulae, commonly used as a semantic representation for feature models. Understanding this concept is crucial to grasp a key step of the approach presented in section 2.4.2.

To express feature model constraints in a propositional formula, we can represent each feature as propositional variables and use traditional Boolean operators ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$)[4]. Let *pFormula* be a function that takes a feature model *FM* as input and returns its representation in propositional formula as output. The output of *pFormula(FM)* is the conjunction of logical representations of each parent-child relationship in *FM*, following the rules outlined in Figure 2.3 [1].

Taking our *Car* example shown in Figure 2.1, we can represent its features using the following propositional variables shown in Table 2.1. Selecting a feature in a configuration is equivalent to setting its corresponding propositional variable to true. Therefore, a configuration is considered valid if the whole formula *pFormula(Car)* is evaluated as true.

Features	Car	Car body	Tran.	Engine	Pulls T.	Manual	Auto.	Elec.	Gas.
Prop var.	C	CB	T	En	PT	M	A	El	G

Table 2.1: Propositional variable translation for the *Car* feature model from Fig. 2.1

Then, *pFormula(Car)* would be: $C \wedge (C \rightarrow CB) \wedge (CB \rightarrow C) \wedge (C \rightarrow T) \wedge (T \rightarrow C) \wedge (T \rightarrow (T \wedge \neg M) \vee (\neg T \wedge M)) \wedge (M \rightarrow T) \wedge (A \rightarrow T) \wedge (C \rightarrow En) \wedge (En \rightarrow C) \wedge (En \rightarrow (El \vee G)) \wedge (El \rightarrow En) \wedge (G \rightarrow El) \wedge (PT \rightarrow C)$.

On a side note, the propositional formula resulting from the translation process is raw and can potentially be simplified.

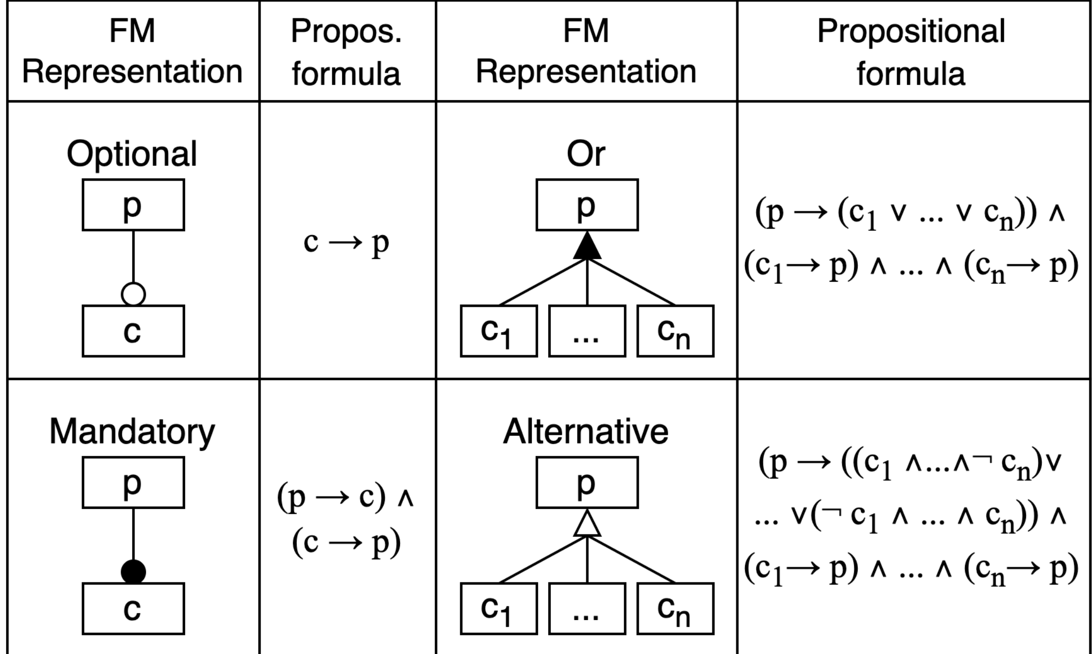


Figure 2.3: Converting feature model constraints to propositional logic.

2.2 Feature-based context-oriented programming

In this section, we aim to provide a brief introduction to the Feature-Based Context-Oriented Programming (FBCOP) approach [8, 10]. We will begin by discussing the reasons behind the emergence of this type of programming paradigm and the specific problems it seeks to address in section 2.2.1. Subsequently, we will delve

into the workings of the FBCOP approach and explore how an FBCOP system can be represented using a Context-Feature-Mapping (CFM) model.

2.2.1 Context-Oriented Programming

Context-Oriented Programming (COP) is a programming paradigm that enables developers to create context-aware systems [3]. These systems inherently adapt their behaviour in response to their environment in which they operate, with context embodying various environmental aspects like user availability or current weather conditions. The idea behind such paradigmn is to facilitate the development of more flexible and adaptive systems whereas traditional programming paradigms often lack modularity and flexibility.

In 2008, Hirschfeld et al. provided a definition of context as "*any information which is computationally accessible may form part of the context upon which behaviour variations depend*" [17]. This concept plays a central role in Context-Oriented Programming. Additionally, Hartmann et al. introduced the notion of mapping between a Context Variability Model and a Feature Model in the context of Software Product Lines research in the same year [13]. Figure 2.4 illustrates this mapping, which serves as a source of inspiration for the FBCOP approach as will be explained in subsection 2.2.2.

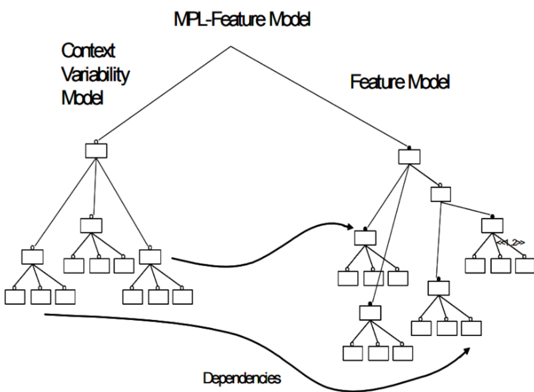


Figure 2.4: MPL-Feature Model Diagram by Hartmann and Trew [13].

2.2.2 Feature-based context-oriented programming approach

This approach builds upon Context-Oriented Programming, Feature Modelling and Dynamic Software Product Lines. It employs two distinct feature models, namely the *Context Model* and the *Feature Model*, to explicitly represent contexts and

features, respectively. The *Context Model* defines the possible contexts and their corresponding constraints that the system can adapt to. It can be seen as the representation of the surrounding environment in which the system operates. On the other hand, the *Feature Model* declares the behavioural variations of the system. On top of that, the dependencies between contexts and features are specified in a *Mapping Model*, enabling the system to identify the features that need to be installed or removed based on the current context. This allows the system to adapt its behaviour to the changing contextual environment at runtime. When the three models are combined, they form a *Context-Feature Mapping* (CFM) model. A detailed explanation and an example of this modelling notation will be provided in section 2.2.3.

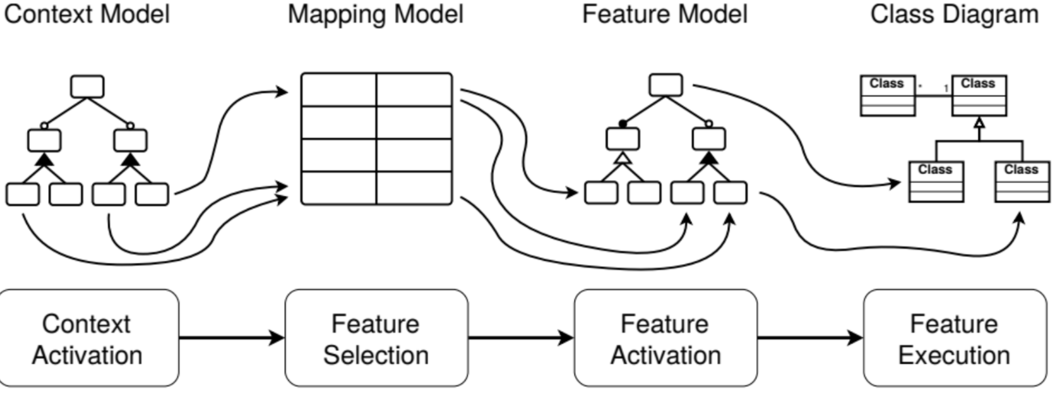


Figure 2.5: Control flow of FBCOP language, extracted from [10].

The control flow of the FBCOP language is unilateral, which implies that the activation of features depends on the context, and not vice versa. Hence, activating a feature cannot trigger the activation of a context.

Figure 2.5 illustrates the control flow of the approach, which works in the following manner: when a context changes, the *Context Activation* component is notified and tries to activate it, taking into account the constraints imposed by the *Context Model.* Next, the *Feature Selection* component selects the features that correspond to the newly activated context(s) based on the *Mapping Model.* The *Feature Activation* component attempts to activate the selected features while also considering the constraints imposed by the *Feature Model.* Finally, the activated features are deployed in the classes of the current system by the Feature Execution component. The same goes for the deactivations of contexts, features, and code.

2.2.3 Context-Feature-Mapping model

As introduced above, Context-Feature-Mapping (CFM) models help to represent the variability within FBCOP systems. This model is comprised of a set of contexts, a set of features, and their relationships represented as a set of mappings. As in feature models, each configuration of the CFM model represents a different configuration of the software system. Figure 2.6 provides an example of a simple CFM model, extracted from our running example representing a smart messaging application that will be introduced in Chapter 3.

A configuration of a CFM model consists of a subset of selected features that includes the root *Feature*, a subset of selected contexts that includes the root *Context*, and a subset of mappings from contexts to features. The configuration is deemed valid if it adheres to all the model's constraints (i.e. constraints shown in Figure 2.2). For example, a valid configuration considering the CFM in Figure 2.6 would be: set of selected contexts = *{Context, UserAvailability, Busy, Peripheral, HasCamera, HasMicrophone}*, set of selected features = *{Feature, Mode, Mute, MessageType, Picture, Vocal, Text}*, set of mapping links = *{Busy:Mute, HasCamera:Picture, HasMicrophone:Vocal}*.

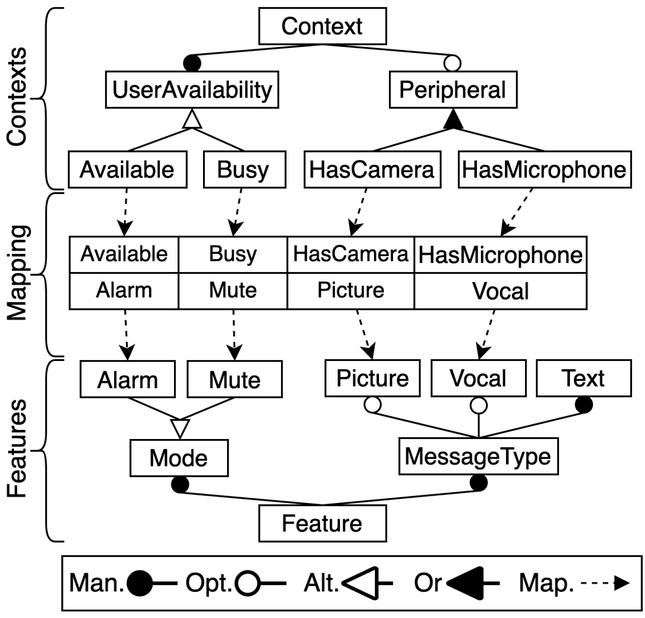


Figure 2.6: Example of a CFM model extracted from our running example representing a smart messaging application introduced in Chapter 3.

2.2.4 Converting CFM model to conjunctive normal form

This section builds upon the concept introduced in section 2.1.3, which discussed the conversion of feature models into propositional formulae. Here, we aim to extend this process to CFM models and take it one step further by converting them into Conjunctive Normal Form (CNF), which allows for the use of SAT solvers. An explanation of what SAT solvers are will be provided in the next subsection 2.3.

As we observed, propositional formulae are frequently employed as a semantic representation for feature models. Likewise, this type of representation can be utilized to represent CFM models. To derive the propositional formula for a CFM model, one would typically take the logical conjunction of the parent-child relationships of each context and feature model. However, this is incomplete. Indeed, one must also consider the mapping part of the CFM model as well.

Taking our simple example shown in Figure 2.6, the links defined by the mapping model can be represented by implications, representing which contexts activates which features. The resulting output of *pFormula(CFM)* is then the conjunction of *pFormula(Feature Model part)*, *pFormula(Context Model part)* and the following implications: *(Available* $\rightarrow$ *Alarm)* $\wedge$ *(Busy* $\rightarrow$ *Mute)* $\wedge$ *(HasCamera* $\rightarrow$ *Picture)* $\wedge$ *(HasMicrophone* $\rightarrow$ *Vocal)*. A more detailed explanation about converting the context-feature mapping part of the CFM model is given by Martou et al. [21].

In our approach, we will require to determine whether a test predicate is evaluated to true or false. This can be achieved by utilizing a SAT solver. However, before using the solver, it is necessary to convert our CFM model into a semantic representation that can be interpreted by the solver. Therefore, it is crucial to transform the CFM model into CNF formulae.

A CNF formula is a way of representing a boolean formula as a conjunction (i.e., a logical *and*) of clauses, where each clause is a disjunction (i.e., a logical *or*) of literals (i.e., variables or their negations). For example the CNF formula of a mandatory parent-child relationship between a parent p and a child c is: $(c \vee \neg p) \wedge (\neg c \vee p)$. This formula contains two clauses over the literals p and c.

The CFM representation of basic parent-child relationship constraints, as well as their CNF conversion, are illustrated in Figure 2.7.

CFM Representation	CNF formula	CFM Representation	CNF formula
Optional p c	$(\neg c \lor p)$	Or p c_1 ... c_n	$(c_1 \lor \ldots \lor c_n \lor \neg p) \land$ $(\land_1^n (c_n \lor \neg p))$
Mandatory p c	$(c \lor \neg p) \land$ $(\neg c \lor p)$	Alternative p c_1 ... c_n	$(c_1 \lor \ldots \lor c_n \lor \neg p) \land$ $(\land_1^n (c_n \lor \neg p)) \land$ $(\land_{i<j} (\neg c_i \lor \neg c_j))$

Figure 2.7: Converting basic CFM model constraints to conjunctive normal form.

2.3 SAT solver

In section 2.1.3, we learned about the conversion of a feature model into a propositional logic formula. Later, in section 2.2.4, we discussed the conversion of these formulas into conjunctive normal form to enable the use of SAT solvers. In this section, we aim to define what a SAT solver is. It has been demonstrated in the literature that such solvers are efficient [22] and widely used with feature models [1, 5] and with context feature mapping models [21]. We will use it in the approach presented in section 2.4.2.

A SAT solver is a program that takes as input a boolean formula in CNF and determines whether there exists an assignment of boolean values (true or false) to the variables in the formula that makes the formula evaluate to true. Such an assignment is called a satisfying assignment, and if there is no such assignment, the formula is said to be unsatisfiable.

For instance, taking the CNF formula $(c \lor \neg p) \land (\neg c \lor p)$ from the last section representing a mandatory parent-child relationship between a parent p and a child c. The following assignment $\{c = true,\ p = true\}$ satisfies the formula.

2.4 Mutation Testing

This final section of our background chapter will cover mutation testing as it is the software testing technique that will be explored in this master thesis. We will first give a short introduction in subsection (2.4.1) by defining mutation testing and explain how it is traditionally performed. Then, in the second subsection (2.4.2), we will describe how existing techniques apply mutation testing to features models.

2.4.1 Traditional mutation testing

Mutation testing has its origins in the 1970s, and has since been employed across various application domains, continuously gaining in popularity. Extensive surveys on the subject can be found in the literature [18, 26].

Mutation testing (or mutation analysis) is defined as a fault–based software testing technique [24], that serves to evaluate the effectiveness of an existing set of test cases in detecting faults or defects in a piece of software. It involves introducing small intentional changes, called *mutations*, into the source code of the software and re-running the test cases to see if they are able to detect the changes.

It is possible to derive syntactic transformation rules, called *mutant operators*, that define how to introduce syntactic changes to the program. For instance, in a simple equation, an arithmetic mutant operator could be changing a $+$ to a $-$, another one could be changing a constant by another. We call *original*, the model we are starting with, and *mutant*, the model obtained after applying a *mutant operator* to the original model.

The idea behind mutation testing is thus simple: create defects and instruct testers to design a test suite capable of detecting them. However, one might ask why such an approach is effective. The effectiveness of this approach relies on two key assumptions: the competent programmer hypothesis and the coupling effect.

The competent programmer hypothesis assumes that competent programmers will produce near-correct programs from a behavioural point of view, they are not programming randomly. As a result, their programs require minimal syntactic modifications to become correct. The second assumption, the coupling effect, suggests that minor errors can combine and result in other unexpected errors. If a test suite can detect a minor error, it is sensitive enough to implicitly distinguish more intricate ones [7].

Let us now demonstrate another advantageous aspect of mutation testing by examining a basic Python program alongside two potential mutants. The first mutant, *m1*, modifies the if condition by changing the operator from "$<$" to "$>$". The second mutant, *m2*, alters the constant a from "*1*" to "*3*". The program will be deemed correct if it presents "ok" in the console.

```python
a = 1
b = 5
if a < b:
    print("ok")
else:
    print("not ok")
```
Orginal program

```python
a = 1
b = 5
if a > b:
    print("ok")
else:
    print("not ok")
```
Mutant m1

```python
a = 3
b = 5
if a < b:
    print("ok")
else:
    print("not ok")
```
Mutant m2

We can see that *m1* breaks the behaviour of our program whereas *m2* does not. If a mutant does not pass a test from our test suite, it is said to be *killed*. If it does, it is said to be *survived*. Detecting killed and survived mutants allow testers to get a mutation score measuring the coverage of the testing suite, given by *#killed/(#killed + #survived)*. The more mutant are killed, the more effective the test suite is.

This enables us to delve into a common problem in software testing, which is determining when to end the testing process. How do we know when the testing suite has been thorough enough? This technological issue reflects a more profound philosophical dilemma known as "*Quis custodiet ipsos custodes?*", or "*Who will guard the guards?*". The mutation testing process provides a solution to this problem by stopping when the mutation score reaches a specified threshold, ensuring a certain level of confidence.

Finally, let us discuss the problem of *equivalent mutants*, which is a significant issue that needs to be addressed. These are mutants that have *survived* the mutation process but do not affect the behaviour of the program. Testers do not want to waste resources testing all possible *equivalent mutants*, nor do they want to create "fake" tests to cover them. In the next subsection, we will introduce an approach that can effectively distinguish between *equivalent mutants* and *mutants*, resolving this issue.

2.4.2 Feature-based mutation testing

In this subsection, we will look at mutation testing from another perspective, which is model-based mutation testing. We will discuss two different approaches for testing software product lines (SPLs), both involving mutations on features models.

A first approach with the aim of generating test configurations for a SPL, proposed by Henard et al. [16, 15], involves converting a feature model into a logical formula, which is then mutated using logical operators. Here, mutations are performed at the level of the logical formula with the help of a tool called MutaLog [14].

In 2015, Arcaini et al. proposed another approach [1, 2] that uses mutations performed directly on the tested feature model, at a human-readable level. This method allows for greater clarity and understanding when generating tests to detect defects in feature models, while also having a means to differentiate between equivalent mutants and mutants. Hence, this will be the method that will be chosen as the starting point of our testing process that will be explained in Chapter 5.

Figure 2.8 illustrates how Arcaini et al. [1, 2] generate a distinguishing configuration between *M* and *M'* after applying a *ManToOpt* mutant operator. A *distinguishing configuration* is defined as a configuration that is valid in an original model *M* but not in its mutant *M'*, or vice versa. In the example shown in Figure 2.8, the following configuration *{a=true, b=false, c=true}* is a distinguishing configuration.

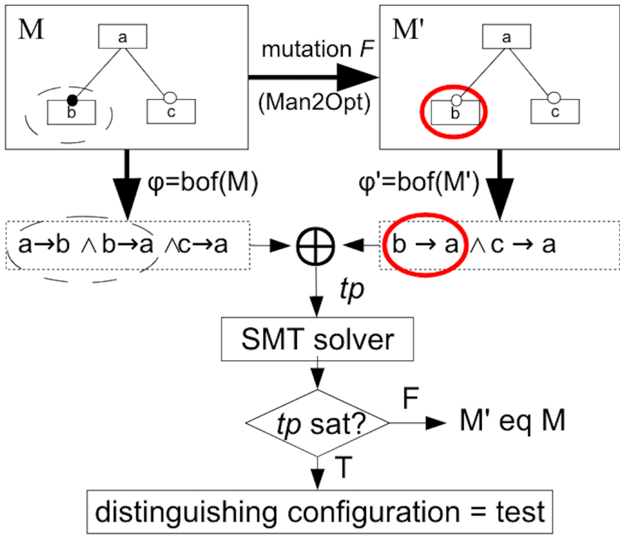


Figure 2.8: Generating a distinguishing configuration between *M* and *M'* after applying a *ManToOpt* mutant operator, extracted from [1].

Once the mutation is done, the two models are transformed into Boolean formulae. Subsequently, a detection condition is created by applying a logical exclusive

(i.e., XOR) between these Boolean formulae. Such condition will be true only if both boolean formuae evaluate to different values. Then they evaluate the detection condition using a logic solver. If a configuration meets the requirements of the detection condition, a distinguishing configuration exists. If not, both *M'* and *M* represent equivalent models.

Several mutant operators have been proposed in literature to conduct feature-based mutation testing [1, 27]. The commonly used ones are **OptToMan**: An Optional is tranformed in Mandatory, **ManToOpt**: A Mandatory is transformed in Optional, **OrToAlt**: An Or is transformed in an Alternative, **OrToOpt**: An Or is tranformed in Optional, **OrToMan**: An Or is transformed in Mandatory, **AltToOr**: An Alternative is transformed in an Or, **AltToOpt**: An Alternative is transformed in Optional, **AltToMan**: An Alternative is transformed in Mandatory. Their representations are shown in Figure 2.9.

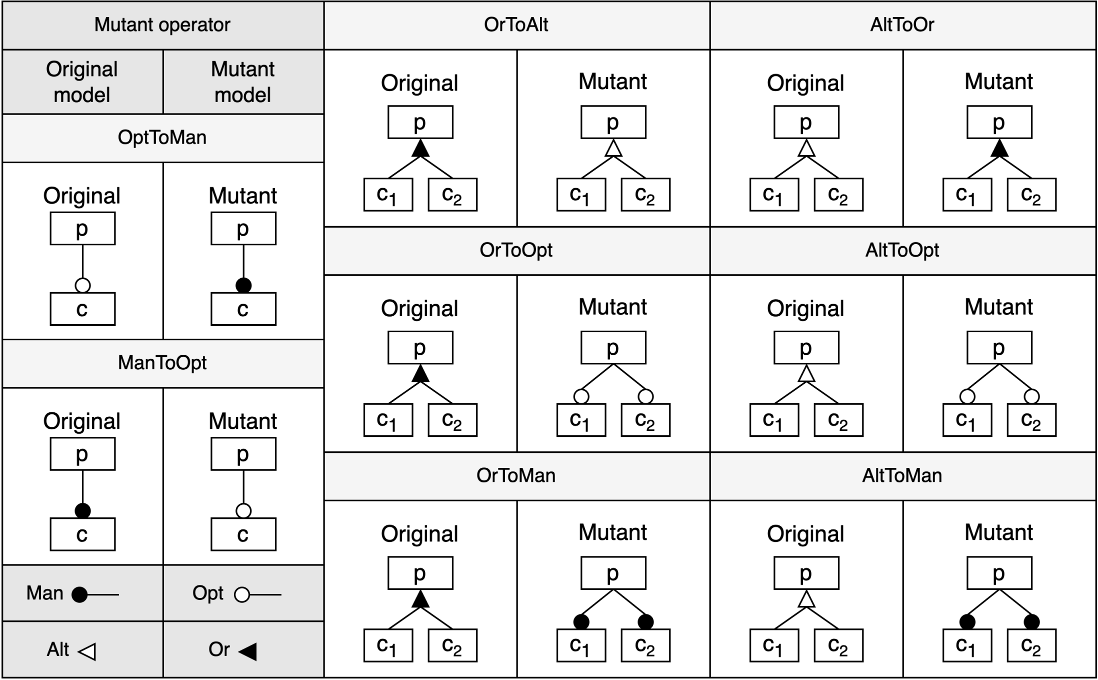


Figure 2.9: Mutant operators to conduct feature-based mutation testing, inspired by [1, 27].

It is worth noting that there exist other mutant operators that deconstruct the hierarchy of the feature model tree, such as repositioning a child feature next to its parent, and so on.

2.5 Background Summary

In the background chapter, we provided a comprehensive overview of the key elements crucial to our master thesis research. We began by examining the concept of feature modelling, a fundamental notation for understanding FBCOP systems. This exploration provided an understanding of how features are represented and organised in software development. We then discussed feature-based context-oriented programming (FBCOP), highlighting its purpose and relevance for addressing specific challenges in software systems. We also introduced SAT solvers, a concept on which our recommendation system is based. Finally, we explored existing mutation testing approaches and their applicability to feature models. With this comprehensive background knowledge in place, we now move on to the next chapter, where a running example will be introduced, which will guide us through the master thesis, providing practical context and enhancing our understanding of the topic.

Chapter 3

Running example

The goal of this chapter is to introduce a smart messaging application, which will be used whenever an illustrative example is needed thorough the master thesis. The term *smart* refers to the ability of our chosen application to automatically adapt its functionality to its operating environment, making it a FBCOP application.

This chapter is structured as follows: section 3.1 presents our smart messaging application. Then, section 3.2 explains each feature, context, and mapping models, providing a comprehensive understanding of the example used for the reader.

3.1 Smart messaging application

The main objective of our application, which is represented in Figure 3.1, is to enable message exchange and contact management among users. Therefore, key features include *Sending*, *Receiving*, and a *ContactSystem*. These features represent the core functionality of the application and are example of its commonalities.

Subsequently, additional features that rely on the context in which they operate can be incorporated, such as the ability to translate messages based on the country the user is located in (*TranslateMessage*). These types of features are not crucial to the functioning of the application and are therefore considered variabilities.

It is worth mentioning that some design faults have been deliberately incorporated into the CFM model in order to assess the ability of our recommendation system to identify them. Also the concept of a smart messaging application was introduced in the course *LINFO2252 - Software maintenance and evolution* given by Prof. Kim Mens at UCLouvain, and my version of it draws inspiration from my work in the course as a student, as well as from the research by Martou et al. [21].

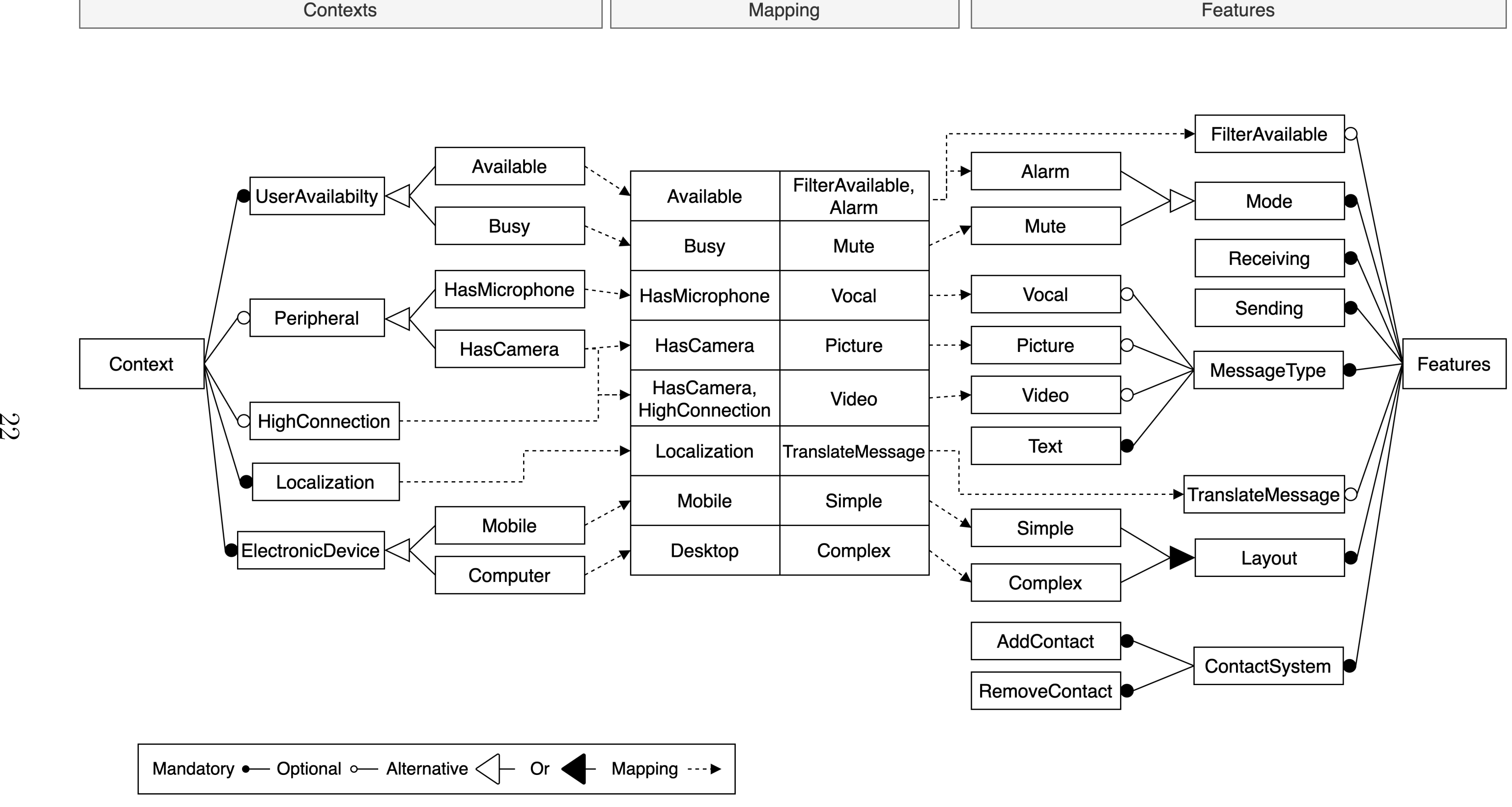


Figure 3.1: Context-feature-mapping (CFM) model representing a smart messaging application.

3.2 Lexicon

The purpose of this section is to explain the meaning of each context, feature and mapping relationship, in order to provide the reader with a detailed understanding of the example used. Even contexts and features that may seem self-evident will be included, to avoid any possible confusion.

3.2.1 Context model

- *UserAvailability* is an abstract context representing the current availability of the user. It includes two child-features: *Available* and *Busy*, which indicate whether the user is currently available or occupied.
- The context *Peripheral* refers to the presence or absence of a camera or microphone attached to the electronic device used by the user. It includes two child-features: *HasCamera* and *HasMicrophone*.
- *HighConnection* is a context activated whenever the connection of the user is high enough to send and watch videos.
- The context *Localization* refers to the geographic location of the user, typically referring to the country where the user is situated.
- The context *ElectronicDevice* specifies the type of the device used by the user to access the application, which can either be a *Mobile* device such as a smartphone or tablet, or a *Computer* device such as a laptop or desktop.

3.2.2 Feature model

- *FilterAvailable* is an optional feature that permits available users to search for other available users for initiating a discussion. It is not considered as an essential component of the core application system.
- *Mode* represents the mode which the device is in. It could be either *Alarm* or *Mute*. When the device is in *Alarm* mode, it can produce sounds, for example when notifications are received. On the other hand, when it is in *Mute* mode, it will remain silent.
- The feature *Receiving* refers to the ability of the user to receive messages.
- *Sending* is a feature allowing users to send messages.

- The feature *MessageType* defines the type of message being sent by the application. *Text* messages are a commonality and are therefore represented by a mandatory parent-child relationship. The other message types, such as *Vocal*, *Picture*, and *Video*, are variabilities and are represented through optional parent-child relationships.

- *TranslateMessage* is an optional functionality that automatically translates the messages sent or received by users.

- The feature *Layout* represents the format in which the application is presented, and it can either be a *Simple* format with fewer functionalities or a *Complex* format with more functionalities.

- *ContactSystem* is a commonality allowing users to *AddContact* or *Remove-Contact*.

3.2.3 Mapping model

- If the user is *Available*, then he will be able to filter for other available users using *FilterAvailable*.

- If the user is *Available*, then his device will be in *Alarm* mode. Otherwise, if he is *Busy*, his device will switch in *Mute* mode.

- If the device does have a microphone (*HasMicrophone*), then *Pictures* can be sent by the application.

- If the device used does have a camera (*HasCamera*), then *Vocals* can be sent by the application.

- If the device does have a camera (*HasCamera*) while having a high connection (*HighConnection*), then *Videos* can be sent by the application.

- If the *Localization* of the user is activated, then he is able to translate his messages (*TranslateMessage*) according to the different languages of the country he is located in.

- The *Layout* of the application will depend on the *ElectronicDevice* context of the user. If the user is accessing the application from a *Mobile* device like a smartphone or a tablet, the *Layout* will be set to *Simple*. On the other hand, if the user is using a *Computer* device such as a laptop or desktop, the *Layout* will be set to *Complex*.

Chapter 4

Motivations, Problem Statement and Analysis

With the background and running example established, we can now delve into the central problem addressed in this master thesis. We will begin in section 4.1 by highlighting the significance of software testing and further explore in section 4.2 the challenges posed by the inherent complexity of FBCOP systems, building upon the issues introduced in the introduction, and mentioning existing solutions in the related work. Then, in section 4.3 we will discuss the potential of mutation testing as a powerful tool for evaluating FBCOP systems. Finally, we will conclude in 4.4 by summarising the key aspects of this chapter.

4.1 The importance of software testing

As previously mentioned in the introduction, testing activities are essential in any software development approach to ensure the quality and reliability of the software. For instance, in our smart messaging application introduced in Chapter 3, a design error could result in an intrusive ringing phone during a meeting, which is a bothersome issue but with less severe consequences than other design errors in applications that manage people's lives. For example, in a risk information system [9], a simple design flaw could lead to catastrophic incidents resulting in unnecessary loss of lives that could have been avoided if the software would have worked properly. Having established the significance of testing in a software development approach, let us now delve into the challenges posed by FBCOP systems and shed light on the difficulties they bring.

4.2 The complexity within FBCOP systems

We mentioned in the introduction that FBCOP systems introduce unique challenges due to the dynamic addition or removal of highly variable behavioural variations. Additionally, the capability of these variations to refine one another results in a complex and dynamic control flow. As a consequence, software testing in such systems becomes intricate, necessitating new specialized approaches.

A first challenge in developing FBCOP systems is effectively managing and keeping track of the multitude of contexts, features, and their intricate activation, dependencies, and relationships. To address this challenge, previous research has introduced a visualization tool that aids developers in navigating and comprehending the inherent complexity of these systems [9, 11, 12].

Another significant challenge lies in ensuring that the interactions between features align with the intended behaviour. Certain combinations of features may produce unexpected behaviour. However, since any configuration within the FBCOP system could potentially exhibit such unexpected behaviour, it is a complex task for a developer to identify which combinations of features may produce unintended behaviours and thus to identify configurations that require specific testing.

To address this challenge, one would generate a comprehensive test suite that encompasses all configurations within the CFM model, effectively preventing unintended behaviours. Nevertheless, this approach introduces its own challenge as the number of interactions between features, contexts, and mapping relationships grows exponentially, making it impractical to test every configuration. To tackle this challenge, previous research has explored the use of pairwise combinatorial interaction testing (CIT) to effectively manage the exponential growth of configurations [21].

A final challenge is to ensure that the CFM model accurately reflects the intended behaviour of the FBCOP system envisioned by the designer. With numerous contexts, features, mappings, and relationships involved, designing a CFM model that precisely captures the system's behaviour is complex and prone to design errors. This master thesis aims to address this challenge by developing a recommendation system. The system will challenge the design choices made in the CFM models and encourage designers to assess whether their choices accurately represent the FBCOP system they have imagined, by making them step back.

4.3 The choice of selective mutation testing

Having discussed the complexity involved in testing FBCOP systems, It is important to discuss the choice of using a selective mutation testing technique to develop our recommendation system. As mentioned earlier in section 2.4, mutation testing has gained significant recognition in the field of software testing and has been successfully applied to diverse systems [26]. There are several aspects that motivated our focus on mutation analysis.

First of all, the very essence of mutations makes them an ideal starting point for the development of a recommendation system. The ability to make small modifications to the designer's CFM model allows us to effectively challenge certain part of the model while preserving its overall behaviour.

Furthermore, given the need to manage the exponential growth of configurations in the FBCOP system being tested, it is crucial to have a powerful tool that not only effectively handles this challenge but also provides a reliable measure of confidence. Previous research on model-based mutation analysis has demonstrated promising results in generating a manageable number of configurations that grows linearly with the number of features in the features models [1, 2, 16]. This aligns with the objective pursued in this master thesis.

On top of that, the use of the concept of distinguishing configurations presented in section 2.4.2 [1] allows us to avoid wrongly challenging the CFM designer by precisely identifying equivalent mutants that do not modify the behaviour of the FBCOP system, which results in time and efficiency gains.

Finally, it has been demonstrated in the literature that selective mutations offer comparable coverage to non-selective mutations while significantly reducing costs [25] and exhibiting good scalability [29], even for complex and large-scale systems.

On a side note, our chosen approach involves performing mutations at a human-readable scale directly on the CFM model representing the FBCOP system. However, other avenues can be explored. Being widely adopted, mutation testing can be applied at various stages of the FBCOP approach. One alternative, for example, is to perform mutations directly in the code of the developed features, following a more traditional mutation testing approach. Another option is to introduce mutations after generating different scenarios using the pairwise CIT approach proposed by Martou et al. [21]. These alternatives present interesting avenues for future research and can be explored to evaluate their effectiveness in FBCOP testing methodologies.

4.4 Overview of the Problem Statement

As in any software development approach, testing plays a crucial role in ensuring the quality of the released software. We first highlighted the challenges of accurately designing CFM models and managing the exponential growth of configurations in FBCOP systems. Then, we presented the rationale for using selective mutation testing as a potential solution to these challenges.

Given this context, the objective of our master thesis is to develop a recommendation system that uses selective mutation analysis to critically evaluate the design choices made during the construction of CFM models. The system aims to provide valuable insights and challenges to the designers, enhancing the overall quality of the CFM models.

Chapter 5

Theoretical Foundations and Approach

With the background information, running example, and problem statement established, we will now delve into the elaboration of our recommendation system. Our objective is to develop a user-friendly and accessible system that does not require advanced knowledge to be used. The designer will simply need to provide their CFM model in the appropriate format and answer to a series of questions that challenge their design choices. Based on their answers, our recommendation system will suggest modifications to the CFM model to improve its overall quality, if necessary.

This chapter provides an overview of the theoretical foundations underlying our recommendation system. We will start by discussing the required input format for our system in section 5.1 and the rationale behind it. Next, we will delve into the derivation of FBCOP-specific mutant operators in section 5.2. We will then explore the generation of useful recommendations to be presented to the user in section 5.3. Finally, we will examine the approach for presenting the mutants to the developer and discuss the reasons behind this decision in section 5.4.

5.1 Formatting the CFM model

Our initial step will involve transforming a CFM model, such as the one representing our smart messaging application shown in Figure 3.1 of Chapter 3, into a compatible input format for our recommendation system.

However, before converting the model, it is essential to mention that the context and feature models within the CFM model are assumed to be free of any classical

feature modelling anomalies [5]. Hence, these models must adhere to the following criteria:

- They should not contain any dead contexts or features, which refers to contexts or features that can never be activated in any configuration of the CFM model.
- They should not be void CFM models, meaning CFM models for which it is not possible to derive any valid configuration (i.e. there is no product that can be derived from this CFM model).

This assumption is justified by the availability of various tools that can perform checks on feature models to identify and prevent such feature modelling anomalies. One notable example is FeatureIDE [20, 28], an Eclipse-based IDE widely used for creating feature models while ensuring their integrity and absence of anomalies.

Now that we have a CFM model free of classical feature modelling anomalies, our next step is to generate three text files that capture the relationships within the CFM model. The first file, named *contexts.txt*, will represent the relationships among the different contexts. The second file, *features.txt*, will depict the relationships among the features. The third file, *mapping.txt* will represent the mapping between contexts and features. For instance, the content of the three corresponding files for our smart messaging application are:

```
Content of contexts.txt:
Context/Mandatory/UserAvailability-Localization-ElectronicDevice
Context/Optional/Peripheral-HighConnection
UserAvailability/Alternative/Available-Busy
Peripheral/Alternative/HasMicrophone-HasCamera
ElectronicDevice/Alternative/Mobile-Computer
```

```
Content of features.txt:
Feature/Mandatory/Mode-Receiving-Sending-MessageType-Layout-ContactSystem
Feature/Optional/FilterAvailable-TranslateMessage
Mode/Alternative/Alarm-Mute
MessageType/Mandatory/Text
MessageType/Optional/Vocal-Picture-Video
Layout/Or/Simple-Complex
ContactSystem/Mandatory/AddContact-RemoveContact
```

```
Content of mapping.txt:
Available-ACTIVATES-Alarm-FilterAvailable
Busy-ACTIVATES-Mute
HasMicrophone-ACTIVATES-Vocal
HasCamera-ACTIVATES-Picture
HasCamera-HighConnection-ACTIVATES-Video
Localization-ACTIVATES-TranslateMessage
Mobile-ACTIVATES-Simple
Computer-ACTIVATES-Complex
```

It is worth noting that the input format of our approach shares similarities with the work of Martou et al. [21]. This deliberate choice is due to the fact that we are testing the same systems, and a method for data extraction has already been established. This enables us to use both testing methodologies without requiring multiple input formats for the user.

5.2 Designing FBCOP-Specific mutant operators

The key aspect of our recommendation system lies in the generation of mutants. As explained in section 2.4, we can define transformation rules known as mutant operators to generate mutants from our CFM models. In our approach, we categorize mutant operators into two distinct types: feature-based mutant operators, which are applied to both context and feature models, and mapping-based mutant operators, which are applied specifically to the mapping part of CFM models.

Figure 2.9 in section 2.4.2 presented a collection of mutant operators proposed in previous research for feature-based mutation testing [1, 27]. In subsection 5.2.1, we will examine the differences between these mutant operators and those used by our recommendation system. Then, in subsection 5.2.2, we will present our new mapping-based mutant operators.

5.2.1 Adapting existing mutant operators to CFM models

The main difference between traditional feature-based and context-feature-based mutant operators is not in the operators themselves, but rather in the way they can be applied. The very nature of CFM models prevents us from using them in a conventional way. Indeed, the link between contexts and features, defined by the mapping model, means that some constraints influence others.

For instance, considering the CFM model from our smart messaging application shown in Chapter 3, performing an *AltToOr* mutation on the *UserAvailability* context would not change the behaviour of our application. Such mutation is represented in Figure 5.1.

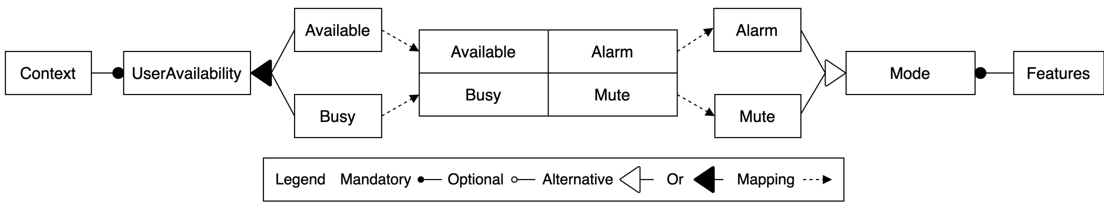


Figure 5.1: Applying an *AltToOr* mutant operator to the *UserAvailability* context.

Indeed, all configurations activating both $\{Available, Busy\}$ contexts at the same time would still be invalid, since $\{Available, Busy\}$ contexts activate both $\{Alarm, Mute\}$ features at the same time. However, the *Mode* feature remains *Alternative*, and by definition, can have exactly one child feature activated between $\{Alarm, Mute\}$ for a configuration to be valid. Thus, the *Alternative* constraint of the *Mode* feature influences the *Or* constraint of the *UserAvailability* context. To address such problem, one would perform an *AltToOr* mutation both on *Mode* and *UserAvailability* at the same time.

However, this process only works if both the *Mode* feature and the *UserAvailability* context are subject to the same constraint (in our case, both are *Alternatives*). Taking now Figure 5.1 as starting point, we encounter a situation where the designer has specified an *Or* constraint on the *UserAvailability* context and an *Alternative* constraint on the *Mode* feature. It is unclear whether the intention was to allow both the *Available* and *Busy* contexts, and thus both the *Alarm* and *Mute* modes, to be activated simultaneously, or not. Therefore, we need to handle both possibilities by performing an *OrToAlt* mutation on the *UserAvailability* context and an *AltToOr* mutation on the *Mode* feature.

In summary, it is crucial to consider both the context and feature that are interconnected through mappings simultaneously when performing mutations, as their constraints can potentially interfere with each other. We will delve later in Chapter 6, into the process of identifying these connected pairs and further explore their implications.

5.2.2 Deriving mapping-based mutant operators

Having introduced the way of adapting traditional feature-based mutant operators to context-feature-mapping models, it is now time to introduce new mutant operators dedicated to the mapping part of the CFM model. To this end, we will introduce new mutant operators from various complexities.

The first one, called *Swap*, is simple and consist of exchanging the mapping relationships between two different contexts and their associated features. Taking our smart messaging application for example, we can imagine a scenario where the designer has wrongly represented the mapping relationships, and has exchanged two of them. Such error is represented in Figure 5.2, where the developer has wrongly mapped the *Available* context to activate the *Mute* feature and the *Busy* context to activate the *Alarm* feature.

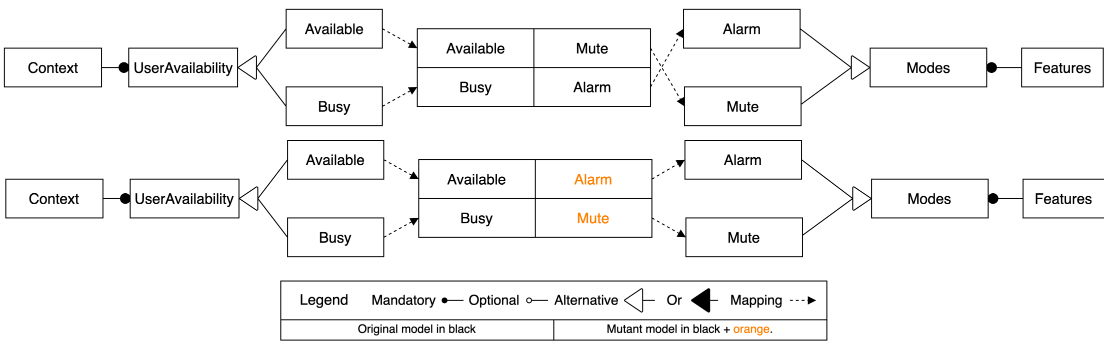


Figure 5.2: Applying a *Swap* mutant operator to the *UserAvailability-Mode* connected pair.

Another one, more intricate, called *Siblings*, helpful to detect whether the designer of the CFM model has forgot some links between features and contexts in the mapping model, is presented in Figure 5.3. In the given model, it is observed that when the *ElectronicDevice* of the user is either a *Smartphone* or a *Computer*, it activates the *Layout* feature, which can be either *Simple* or *Complex*. Additionally, the *Tablet* child of the *ElectronicDevice* context activates the *Complex* child of the *Layout* feature only if its *Peripheral* contains a *VideoCard* context. Otherwise, no *Layout* is activated. Based on this understanding, it can be expected that for each child of *ElectronicDevice*, a *Layout* would be activated. Therefore, when the *VideoCard* context is deactivated and the *Tablet* is activated, it would be expected to activate the *Simple* child of the *Layout* feature, which is not the case.

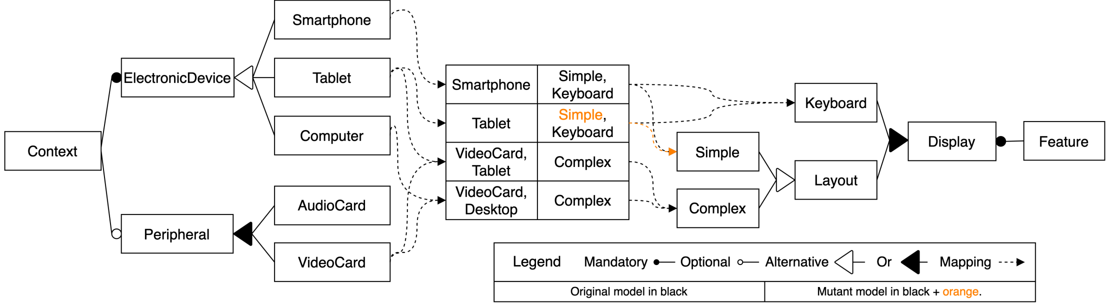


Figure 5.3: Applying a *Siblings* mutant operator to the *ElectronicDevice-Layout* connected pair. Model inspired from [21].

The mechanism of our *Siblings* mutant operator can be summarized as follows:

- It starts with a specific child context (e.g. Tablet).
- It examines the features activated by the other children (i.e., siblings) of its parent context (e.g. *Smartphone* and *Computer*).
- It identifies that each of the other children activates a particular feature (e.g. *Layout*), while the current context does not.
- It modifies the current context to activate the feature that is activated by the other siblings (e.g. add *Simple* feature activation).

Finally, it is worth noting that there are several other mapping-based mutant operators that can be derived to delve deeper into the relationship between context and feature mappings. These operators involve manipulations such as additions, deletions, and modifications of the mapping relationships. However, within this subsection, we have introduced two mapping-based mutant operators that we believe will be particularly valuable in the context of a recommendation system.

5.3 Generating useful recommendations

To effectively challenge the design choices of a developer in constructing a CFM model for an FBCOP system, our system aims to provide recommendations that are as relevant as possible to avoid wasting the time of the developer. This involves generating a meaningful and manageable set of mutants and presenting them to the developer in a prioritized manner, starting with the most promising options and gradually progressing to the less promising ones.

5.3.1 Preventing the generation of equivalent mutants

Avoiding the generation of equivalent mutants (i.e. a mutant that does not modify the system from a behavioural view point) is a crucial and well-recognized challenge in mutation testing. As our objective is to improve the quality of the CFM model by accurately representing the FBCOP system envisioned by the designer, we want to avoid proposing recommendations that do not bring about any meaningful modifications. To address this issue, we will use the concept of distinguishing configurations, as introduced in section 2.4.2 of our background chapter. This concept can be employed in two distinct ways to effectively tackle the challenge at hand.

Firstly, for more intricate mutant operators, we can easily adapt the approach proposed by Arcaini et al. [1] to generate distinguishing configurations within CFM models. This process follows the same explanations as in section 2.4.2 and is illustrated in Figure 5.4.

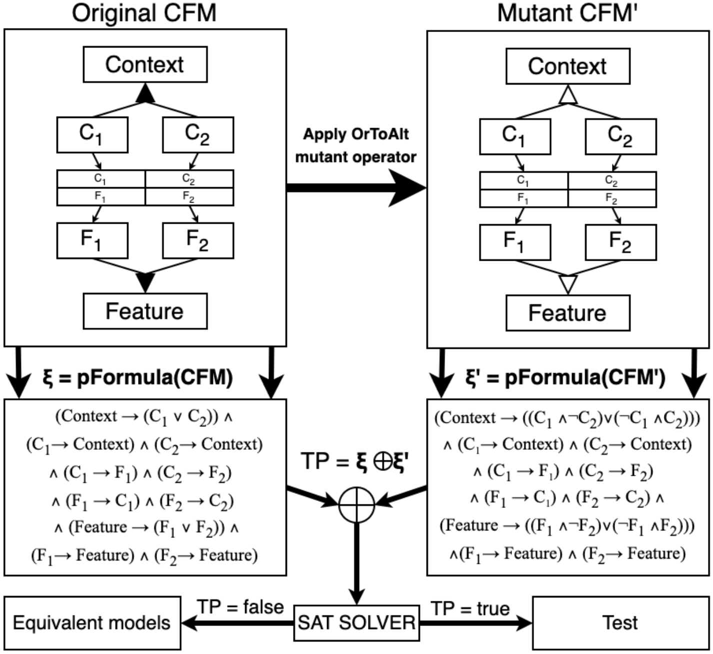


Figure 5.4: Process of generating a distinguishing configuration for an *OrToAlt* mutant operator. Adapted to CFM models and inspired from [1].

However, by leveraging our understanding of FBCOP systems and CFM model constraints, we are able to optimize this process by carefully selecting and focusing on specific mutant operators, that we know will provide valuable insights. This allows us to focus on the most informative and relevant scenarios.

For instance, let us consider our small example extracted from our smart messaging application depicted in Figure 5.1. By carefully examining this mutant operator, we can observe that the distinguishing configuration between an *Or* constraint and an *Alternative* constraint is the activation of multiple children of the parent context or feature. In our specific case, this corresponds to the simultaneous activation of both *Available* and *Busy* contexts.

5.3.2 Introducing constraint hierarchy in CFM models

Having successfully implemented a method to prevent the generation of equivalent mutants in our recommendation system, we can now take it a step further by incorporating the competent programmer hypothesis and introducing a constraint hierarchy in our CFM models. The competent programmer hypothesis, as discussed in section 2.4 of the background, assumes that the developer has captured the intended constraints with reasonable accuracy. By leveraging this hypothesis, we can introduce a constraint hierarchy to reduce the number of generated mutants.

Therefore, our recommendation system is based on the assumption that if a developer has implemented an *Alternative* constraint, the *AltToOr* mutant operator is more likely to be relevant compared to *AltToOpt* or *AltToMan* operators. This is because transitioning from a situation where exactly one child must be activated (Alternative) to a situation where at least one child must be activated (Or) is a smaller leap compared to situations where no children can be activated (Optional) or all children must be activated (Mandatory). Based on this reasoning, we have established a constraint hierarchy, illustrated in Figure 5.5, which outlines the hierarchy of our six selected mutant operators, used in our recommendation system.

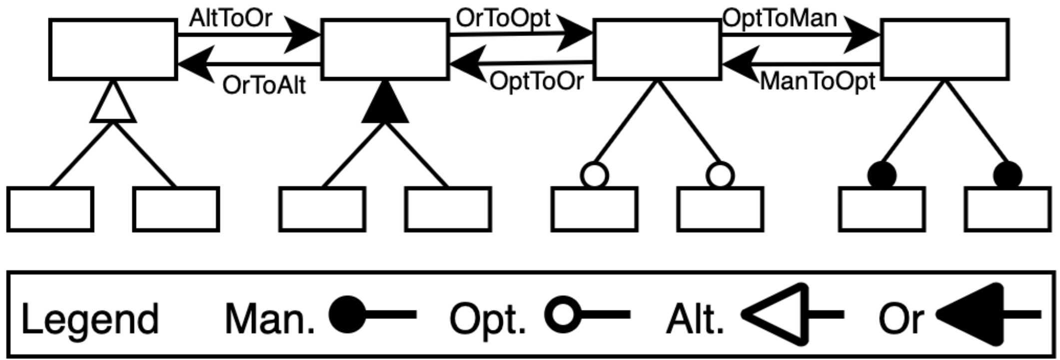


Figure 5.5: Constraint hierarchy, based on the competent programmer hypothesis.

In any case, if the developer has completely misrepresented a constraint in their CFM model, we can safely assume that simply asking a question about it will prompt them to reconsider their model. In addition, we can apply our process iteratively, guiding them to the correct model through several iterations, if necessary.

5.3.3 Prioritizing mutants for presentation

After successfully avoiding equivalent mutants, and reducing the number of mutants to be generated, we believe that certain mutants should be presented to the developer before others in order to maximize the efficiency of our recommendation system.

Therefore, after careful consideration, we have determined that the context-feature-based mutant operators are theoretically the most significant ones to present to the developer. We believe that these specific operators have the greatest potential to provide valuable insights and recommendations.

Subsequently, we will present recommendations based on the sibling mutant operator, followed by those derived from the swap mutant operator. Our rationale for this order is based on the theoretically relatively lower frequency of faults addressed by the sibling mutant operator and the even rarer occurrence of mapping relation misrepresentation by the developer.

However, to provide proper justification for this choice of prioritization, it would be beneficial to consider incorporating user feedback and preferences into the prioritization process, and to test them in real cases. On top of that, we can consider adding various metrics to prioritize mutants, such as the depth in the model or the number of affected features and/or contexts by the mutation.

5.4 Mapping mutant operators to questions

Once we have generated a manageable set of meaningful mutants, our subsequent focus is on outlining the approach for presenting these mutants to the developer. Instead of directly displaying the mutants themselves, we propose a more efficient method that involves asking relevant questions regarding their CFM models. By associating each mutant operator with a specific design question, based on the associated distinguishing configuration, we can effectively challenge the design decisions made by the developer and encourage more reflection.

To illustrate this further, let us revisit our smart messaging application example. As discussed in section 5.3.1, we identified that a distinguishing configuration between an *Alternative* constraint and an *Or* constraint is the activation of multiple children of the parent context or feature. Based on this finding, a relevant question that can be asked is: "*Is it possible for both the Available and Busy contexts to be activated simultaneously?*". If the answer is yes, then it means that the Alternative constraint should be modified in an Or constraint, and remains Alternative if the answer is no.

By following this reasoning, we can derive the following associated questions for each of our mutant operators, where A and B can be either features or contexts:

- AltToOr: "*Is it possible for both A and B to be activated simultaneously?*" If the answer is yes, perform the mutation.
- OrToAlt: "*Is it possible for both A and B to be activated simultaneously?*" If the answer is no, perform the mutation.
- OrToOpt: "*Is it possible for both A and B to be deactivated simultaneously?*" If the answer is yes, perform the mutation.
- OptToOr: "*Is it possible for both A and B to be deactivated simultaneously?*" If the answer is no, perform the mutation.
- OptToMan: "*Do A and B have to be activated in any configuration?*" If the answer is yes, perform the mutation.
- ManToOpt: "*Do A and B have to be activated in any configuration?*" If the answer is no, perform the mutation.
- Siblings: "*Should context A activates feature B?*" If the answer is yes, perform the mutation.
- Swap: "*Does A and B activates the right features?*" If the answer is no, perform the mutation.

5.5 Overview of the different concepts

In this chapter, we have first discussed the chosen input format and the reasons behind its selection. We then introduced our FBCOP-specific mutant operators, classifying them into feature-based and mapping-based categories. Next, we outlined our approach for generating valuable recommendations to the user. Finally, we mapped the introduced mutant operators to design questions, aiming to stimulate thoughtful consideration and critical evaluation of the developer's design choices.

Chapter 6

Recommendation system

After providing an overview of the essential concepts and theoretical foundations of our recommendation system, we will now delve into its mechanism. Section 6.1 will shed light on the architecture of the system that we propose and the interplay between its modules. Following that, in section 6.2, we will present the results obtained using an initial version of the system, implemented in *Python*, and with our smart messaging application example discussed in Chapter 3 as input.

6.1 Recommendation system architecture

The architecture of the recommendation system that we propose is illustrated in Figure 6.1. This diagram provides a visual representation of the various modules and their interactions, showcasing the flow of data exchanges, actions, and interpretations that take place within our recommendation system. Each different module will be described in the next subsections, from the data acquisition and processing to the mutant Q&A validation modules.

It is worth mentioning that the interactions of the developer in the system are limited to two key tasks. Firstly, the developer provides a context-feature-mapping model in the correct format, ensuring its compatibility with the system. Secondly, the developer answers to questions generated through mutation analysis. Based on these questions and their assessment of the CFM model, the developer has the option to make adjustments or adaptations, taking into consideration the suggestions provided by our system or choosing to proceed independently. These interactions are in line with our objective of proposing a simple and comprehensive approach for testing the design of a CFM model.

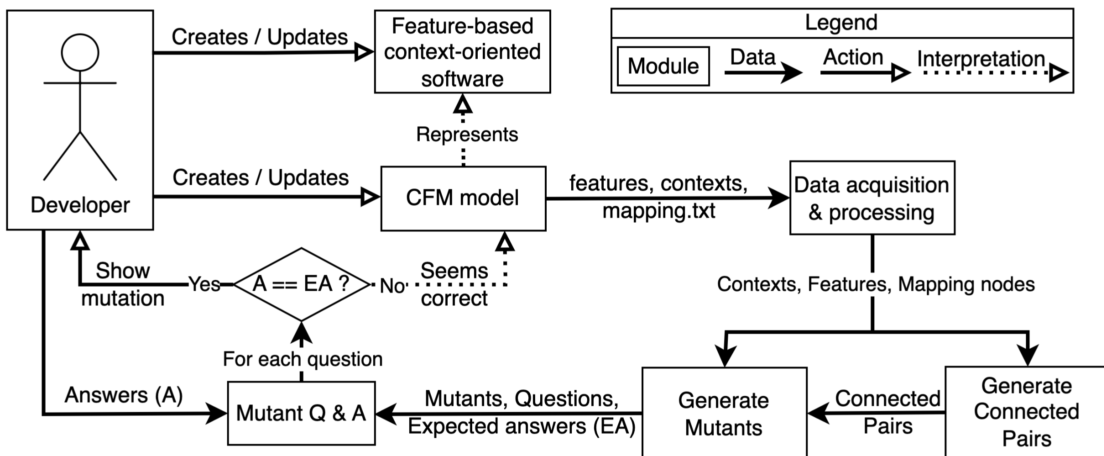


Figure 6.1: Schematic representation of the data exchanges, actions and interpretations between the different modules within our recommendation system.

6.1.1 Data acquisition and processing module

Once the developer has completed the design of their initial context-feature-mapping model, ensuring it is devoid of any conventional feature anomalies, they can begin using our recommendation system. To initiate the process, the first step involves deriving the three input files, namely *contexts.txt*, *features.txt*, and *mapping.txt*, as explained in section 5.1. These files collectively represent the CFM model that will be subjected to testing and evaluation within the system.

Then, the first module will undertake the processing of the input files, extracting and organizing the relevant information. As part of this process, it will instantiate the *CFMmodel* class, which serves as a specialized data structure representing the CFM model being designed by the developer.

The *CFMmodel* class encompasses the following information:

- *CFMnodes*: Represents a relationship as described in the input files, which involves the parent element (a context or a feature), the constraint associated with the relationship, and the corresponding children elements.
- *CFMcontexts*: Refers to CFMnodes which are exclusively representing contexts.
- *CFMfeatures*: Refers to CFMnodes which are exclusively representing features.
- *dictContext*: A dictionary capturing which contexts activate which features.

- *dictFeature*: A dictionary capturing which features are activated by which contexts.
- *connectedPairs*: List containing all the connected pairs.
- *mutants*: List containing all mutants.
- *questions*: List of questions that can be ask to the developer to properly challenge their design choices, alongside their expected answers, and mutations to be performed.

By processing and storing the data in the *CFMmodel* class, the Data Acquisition and Processing module establishes a solid foundation for subsequent modules in the recommendation system.

As an additional and important aspect, this module also carries out automatic checks to ensure the consistency of relationships captured in the *contexts.txt* and *features.txt* files, as well as the mapping connections. These checks are performed to validate and verify the integrity and coherence of the captured data within the system. For instance, it must be ensured that all contexts and features mentioned in the *mapping.txt* files are also mentioned in the *features.txt* or *context.txt* files.

6.1.2 Connected pairs generation module

With the successful instantiation of our *CFMmodel* class and the completion of data processing, the subsequent task for this second module is to leverage this data to identify and extract the connected pairs present within the CFM model. As explained in section 5.2.1, a connected pair refers to a context and a feature that are intricately linked through the mapping dictionaries established in the preceding module, so that their constraints may interfere with each other.

The module begins by accessing the CFM model and examining the mapping dictionaries that define the relationships between contexts and features. It systematically analyzes these mappings to identify instances where a context and a feature are connected. A connected pair is formed whenever at least one child of a context activates at least one child of another feature. In such cases, both the context and the feature are considered to be interconnected and form a connected pair within the system.

Once a connected pair is identified, it is extracted and stored for further processing in subsequent modules. The module continues to iterate through the CFM model, exploring all possible combinations of contexts and features to discover additional connected pairs.

6.1.3 Mutants generation module

Now we delve into the central module of our system, which is responsible for generating mutants through selective mutation testing. In this third module, we traverse the connected pairs and the CFM nodes to determine the appropriate mutant operator to apply. The selection process for mutants in this module adheres to the constraint hierarchy outlined in section 5.3.2. By following this hierarchy, we ensure the generation of a concise and meaningful set of relevant mutants.

Furthermore, this module assumes the responsibility of establishing the mapping between the generated mutants and the questions asked to the designer, as outlined in section 5.4. To achieve this, we maintain a list of questions that contains the actual question itself, the expected answer, and the corresponding mutation associated with the question.

On a side note, it is worth noting that it is possible to extract a sub-model from the generated mutants. This sub-model can enhance the efficiency of the recommendation system by allowing the processing of the same connected pair without the need to recalculate all the nodes or generate mutants for each connected pair repeatedly. By extracting a relevant sub-model, we can streamline the processing and testing procedures, reducing redundancy and optimizing the overall performance of the recommendation system.

6.1.4 Mutant Q&A validation module

The last module of our system is the Mutant Q&A Validation module and acts as the interface through which the developer interacts with the system. Its primary role is to validate the generated mutants and provide feedback to the developer. The module achieves this by sequentially iterating over the list of questions generated by the preceding module.

For each question in the list, the module presents it to the developer and awaits their answer. If the answer of the developer aligns with the expected answer provided in the questions list, the corresponding mutation is suggested to the developer. Conversely, if the answer of the developer does not align with the expected answer, the corresponding mutation is not recommended, indicating that the part of the CFM model currently challenged by the generated mutation accurately reflects the context-oriented software envisioned by the developer.

6.2 Results of our Prototype

To demonstrate the viability of our recommendation system, we have implemented an initial version using the programming language *Python*. All the necessary artifacts, including code and documentation, are accessible on a GitHub repository [6]. Detailed instructions on how to reproduce the results presented in this section are also provided.

6.2.1 Raw results

In section 5.1, we previously introduced the three input files that represent our running example. Building upon these files, the system has generated the corresponding output, which is depicted in Figure 6.2. We will delve into the analysis and discussion of these results in subsection 6.2.2.

```
    The recommendation system will start,
    answer each question carefully. Accepted
    answers are among (yes, y, no, n).

Question 1:
==========
Is it possible for Available,Busy contexts and for Alarm,Mute features to be activated simultaneously? (yes/no):
no

Question 2:
==========
Is it possible for HasMicrophone,HasCamera contexts to be activated simultaneously? (yes/no):
yes
Suggestion: Modify the constraint of Peripheral context from Alternative to Or constraint

Question 3:
==========
Do Localization context(s) have to be activated in any configuration? (yes/no):
no
Suggestion: Modify the constraint of Localization context from Mandatory to Optional constraint

Question 4:
==========
Is it possible for Mobile,Computer contexts to be activated simultaneously? (yes/no):
no

Question 5:
==========
Is it possible for Simple,Complex features to be activated simultaneously? (yes/no):
no
Suggestion: Modify the constraint of Layout feature from Or to Alternative constraint

Question 6:
==========
Is it possible for Available,Busy contexts to be activated simultaneously? (yes/no):
no

             The process is now over.
             Mutation score: 3/6.

Summary of the recommendations:
- Modify the constraint of Localization context from Mandatory to Optional constraint
- Modify the constraint of Peripheral context from Alternative to Or constraint
- Modify the constraint of Layout feature from Or to Alternative constraint
```

Figure 6.2: Output and results of the initial version of the recommendation system on the smart messaging application.

6.2.2 Discussion and analysis

As discussed in Chapter 3, intentional design faults were deliberately introduced into the CFM model representing our smart messaging application. Now, let us assume the role of the developer and evaluate our ability to detect these faults. Within our smart messaging application, there are 11 contexts, 18 features, 8 mapping links and six connected pairs which are as follows: *UserAvailability-Mode*, *UserAvailability-FilterAvailable*, *Peripheral-MessageType*, *HighConnection-Video*, *Localization-TranslateMessage*, and *ElectronicDevice-Layout*. Figure 6.2 illustrates the generation of six questions by our system. Here are the corresponding answers we provided:

- For the first question, we answered no, as we believe it is not possible for a user to be both *Available* and *Busy* simultaneously, nor for the app to be in both *Alarm* and *Mute* mode concurrently.
- Regarding the second question, we answered yes, acknowledging that our device may have both a microphone and a camera attached to it simultaneously.
- We responded negatively to the third question, as we assume that our *Localization* can be deactivated.
- For the fourth question, our answer was no, indicating that we are either connected through a *Mobile* or a *Computer*, but not both simultaneously.
- Similarly, we answered no to the fifth question, stating that only one *Layout* can be activated at a time.
- Lastly, we answered no to the sixth question, which aligns with our response to the first question.

Upon responding to all the questions, we have received multiple suggestions (in our case, three) that can enhance our CFM model, thereby improving its representation of the FBCOP system envisioned by the developer. These suggestions serve as valuable recommendations for refining our CFM model, aligning it more closely with our intended design and enhancing its ability to accurately capture the desired behaviour and functionality of the FBCOP system being represented.

The initial version of our recommendation system has yielded interesting results that align with the objectives outlined in Chapter 4. Through the effective generation of relevant questions to the developer, we have successfully detected all intentional design faults introduced in Chapter 3. It is worth mentioning that, at this stage, our system exclusively supports context-feature-mapping model-based mutant operators, as described in section 5.2.1. Nevertheless, this first version has shown promising outcomes, highlighting its potential for further development and expansion.

Chapter 7

Validation

Having presented the theoretical foundations, the architecture, and a practical demonstration of our recommendation system, the next step is to conduct a short validation analysis. Whereas we have already shown promising results by presenting an initial version of our recommendation system in section 6.2. In this Chapter, we aim to assess whether the system achieves the objectives outlined in Chapter 4 and to identify any limitations or areas for improvement.

7.1 Validation methodology

In order to assess the effectiveness of our recommendation system, we will subject it to a range of CFM models with varying characteristics (for example, a small model, a large model, etc.). This will allow us to evaluate its performance and ability to handle different inputs. We will measure several key metrics, including the number of contexts(Cont.), features(Feat.), mapping links(Map.), connected pairs(C. Pairs), mutants, and questions generated by the system. Additionally, we will record the average processing time for each model, measured in milliseconds, based on 50 runs.

7.2 Raw Results

The validation results, as depicted in Table 7.1, were obtained using the aforementioned methodology. For those interested in replicating the results, the CFM models used for validation, as well as the corresponding scripts, are available on the GitHub repository [6].

Model name	Cont.	Feat.	Map.	C.Pairs	Mutants	Questions	Time[ms]
Single pair	3	3	2	1	2	2	0.685
Small	6	7	4	2	3	3	0.801
Running ex.	11	18	8	6	7	6	1.217
Model1	20	27	14	11	15	15	1.801
Model2	25	32	13	10	11	11	1.885
Big	60	60	35	48	88	88	5.727

Table 7.1: Raw results of our recommendation system applied to a range of CFM models with varying characteristics.

7.3 Discussion and analysis

The results displayed in Table 7.1 prompt a discussion. We will examine them in this section, starting by discussing the efficiency and rapidity of the recommendation system to generate the questions in section 7.3.1, and further discussing the practicality of our system in section 7.3.2.

7.3.1 Efficiency and rapidity

Firstly, our initial version of the recommendation system demonstrates a linear scalability with the number of contexts and features in the CFM model. This means that the number of mutants and questions generated by the system increases proportionally as the number of contexts and features grows. More specifically, it grows proportionally as the number of connected pairs grows.

Secondly, the generation of this set of mutants is achieved efficiently, with a short processing time. For instance, in a model containing 120 nodes, the system completes the generation within approximately 5 milliseconds. This indicates that the system is capable of handling larger CFM models while maintaining a fast processing speed.

On top of that, it is worth noting that the results presented in Table 7.1 do not consider the possibility of reusing a connected pair to extract a sub-model, as discussed in subsection 6.1.3. By implementing this approach and considering the extraction of sub-models, the system would not only achieve efficiency gains but also reduce the processing time required for generating mutants and questions, especially in larger models.

7.3.2 Ease of use

The system is designed to be user-friendly for developers, regardless of their level of expertise in mutation testing, connected pairs, or context-oriented programming. It is accessible and beneficial for both beginners and experts, as it encourages developers to take a fresh perspective on their design approach. The interactions with the system are streamlined and practical, requiring users to provide input in the correct format and answer to questions.

7.4 Threats to validity

Since the models used for validation were manually generated, it is uncertain how our initial version of the recommendation system would perform in cases that are not explicitly represented in these models.

Furthermore, we have currently implemented 6 mutant operators, as discussed in section 5.2.1. To enhance the effectiveness of the system, it would be beneficial to incorporate more mutant operators such as the mapping-based mutant operators discussed in section 5.2.2. However, this expansion would result in generating more questions, necessitating a robust prioritization process to present only relevant recommendations to the user. As our initial version does not include such prioritization, it will be important to reassess the validation process when incorporating these changes into the recommendation system.

7.5 Limitations

Our current implementation focuses only on connected pairs for generating questions. This approach may overlook design errors related to features or contexts that are not explicitly linked in the mapping. For instance, if there is an error in a commonality of the CFM model, such as the *ContactSystem* feature in our smart messaging application, it may go undetected.

On top of that, as we are relying on the answers of the developer, there is a potential for receiving unexpected answers. Although, even in such cases, our system remains aligned with its objectives, since it ultimately reflects the model envisaged and actively designed by the developer, this highlights a limitation of design testing, since the vision of the developer itself may also be subject to inaccuracies or errors.

Chapter 8

Future Work

In this chapter, we discuss potential areas for future research and development that can build upon the findings and contributions of this master thesis. While our current work has provided valuable insights and solutions, there are still several avenues to explore and improve upon. We identified three potential avenues for future work.

8.1 Centralized testing application for feature-based context-oriented programming systems

We intentionally designed our input format to align with existing testing methodologies in the field of FBCOP testing, particularly drawing inspiration from the work of Martou et al. [21]. Additionally, we assume that the input to our recommendation system is free from any common feature anomalies, which can be ensured using tools like FeatureIDE.

One potential avenue for future work is the development of an all-in-one tool that integrates such testing methodologies and tools into a centralized platform. This would enhance user convenience by providing a better interface and a comprehensive solution for FBCOP testing, through different methodologies. Modern frameworks like React.js could be employed to build such a tool, ensuring a user-friendly and efficient interface. By consolidating different tools and methodologies, developers would have a unified and streamlined testing experience.

8.2 Pushing further our recommendation system

Another potential avenue for future work involves expanding the list of mutant operators and automating the process of generating distinguishing configurations. While our current set of mutant operators serves as a starting point, further exploration and inclusion of additional operators could enrich the recommendation system. This would require a systematic analysis of different contextual, feature and mapping-related scenarios to identify new types of mutants that could provide valuable insights.

Additionally, to enhance the prioritization process, it would be beneficial to base it on real use cases or concrete criteria. This could involve gathering user feedback, conducting empirical studies, or defining objective metrics to rank the relevance and impact of different mutants. In the initial version of our system outlined in Chapter 6, we keep track of various metrics such as the depth of the context and feature within their respective models. However, we do not currently leverage these metrics in the functionality of the system. By incorporating real data and more rigorous criteria, the prioritization process would become more robust and effective in guiding the design choices of the developer, while allowing us to add more mutant operators.

8.3 Investigate other mutation testing locations

Let us recall that our approach focuses on selective mutation testing within the context of CFM models. By employing this specific type of mutation testing, we aim to generate mutations at a human-readable scale. The objective is to thoroughly test the design of CFM models and ensure that they accurately represent the envisioned FBCOP system as intended by the developer.

However, it is important to acknowledge that this application is one among many possible areas where mutation testing can be highly effective. We firmly believe that mutation testing has the potential to yield valuable insights and benefits in various other contexts as well, considering FBCOP systems. For instance, a first location could be to apply mutations testing directly in the code of the FBCOP systems under development, (e.g. to the developed features).

Chapter 9

Conclusion

The primary focus of this master thesis was to develop a recommendation system based on mutation analysis to challenge the design choices made by developers when constructing a context-feature-mapping model for a feature-based context-oriented application. The goal was to effectively manage the complex control flow and handle the potential explosion of configurations within the system.

To achieve this objective, we introduced an architecture for a recommendation system designed to generate a concise set of relevant mutants. These mutants were then mapped to a list of targeted questions that we asked to the developer. The efficiency of our approach was demonstrated by applying an initial version of the recommendation system to a smart messaging application. The results clearly indicated that we were able to effectively identify errors within the application in a efficient manner.

Furthermore, our study highlighted the significant role that mutation testing can play in the realm of context-oriented software testing, with potential applications at different stages. Additionally, we discovered potential avenues for enhancing our recommendation system. One such area involves expanding the range of mutant operators used in the process. By introducing new operators, we can offer more diverse and comprehensive recommendations to the developer. Another area is the elaboration of a robust prioritization process when asking questions to the developer. This would ensure that the most relevant and impactful questions are asked first, optimizing the effectiveness of the recommendation system.

In conclusion, this master thesis has generated promising results that warrant further investigation and exploration in future studies.

Bibliography

[1] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. “Generating tests for detecting faults in feature models”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[2] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. “Automatic detection and removal of conformance faults in feature models”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2016, pp. 102–112.

[3] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. “A survey on context-aware systems”. In: *International Journal of Ad Hoc and Ubiquitous Computing* 2.4 (2007), pp. 263–277.

[4] Don Batory. “Feature models, grammars, and propositional formulas”. In: *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005. Proceedings 9*. Springer. 2005, pp. 7–20.

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. “Automated analysis of feature models 20 years later: A literature review”. In: *Information systems* 35.6 (2010), pp. 615–636.

[6] A. Deckers. *TFEmutaCOP*. `https://github.com/audricdec/TFEmutaCOP`. 2023.

[7] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. “Hints on test data selection: Help for the practicing programmer”. In: *Computer* 11.4 (1978), pp. 34–41.

[8] Benoît Duhoux. “Feature-Based Context-Oriented Software Development”. PhD thesis. UCLouvain, 2022.

[9] Benoît Duhoux, Kim Mens, and Bruno Dumas. “Feature visualiser: An inspection tool for context-oriented programmers”. In: *Proceedings of the 10th ACM International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*. 2018, pp. 15–22.

[10] Benoît Duhoux, Kim Mens, and Bruno Dumas. "Implementation of a feature-based context-oriented programming language". In: *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming*. 2019, pp. 9–16.

[11] Benoît Duhoux et al. "A Context and Feature Visualisation Tool for a Feature-Based Context-Oriented Programming Language." In: *SATToSE*. 2019, pp. 1–12.

[12] Benoît Duhoux et al. "Dynamic visualisation of features and contexts for context-oriented programmers". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2019, pp. 1–6.

[13] Herman Hartmann and Tim Trew. "Using feature diagrams with context variability to model multiple product lines for software supply chains". In: *2008 12th International Software Product Line Conference*. IEEE. 2008, pp. 12–21.

[14] Christopher Henard, Mike Papadakis, and Yves Le Traon. "MutaLog: A Tool for Mutating Logic Formulas". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE. OH, USA, 2014, pp. 399–404. DOI: 10.1109/ICSTW.2014.54.

[15] Christopher Henard, Mike Papadakis, and Yves Le Traon. "Mutation-based generation of software product line test configurations". In: *Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings 6*. Springer. 2014, pp. 92–106.

[16] Christopher Henard et al. "Assessing software product line testing via model-based mutation: An application to similarity testing". In: *2013 IEEE Sixth international conference on software testing, verification and validation workshops*. IEEE. 2013, pp. 188–197.

[17] Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. "Context-oriented programming". In: *Journal of Object technology* 7.3 (2008), pp. 125–151.

[18] Yue Jia and Mark Harman. "An analysis and survey of the development of mutation testing". In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[19] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[20] Thomas Leich et al. "Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach". In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 2005, pp. 55–59.

[21] Pierre Martou et al. “Test scenario generation for feature-based context-oriented software systems”. In: *Journal of Systems and Software* 197 (2023), p. 111570.

[22] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. “SAT-based analysis of feature models is easy”. In: *Proceedings of the 13th International Software Product Line Conference*. 2009, pp. 231–240.

[23] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. “A context-oriented software architecture”. In: *Proceedings of the 8th ACM International Workshop on Context-Oriented Programming*. 2016, pp. 7–12.

[24] Larry J. Morell. “A theory of fault-based testing”. In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 844–857.

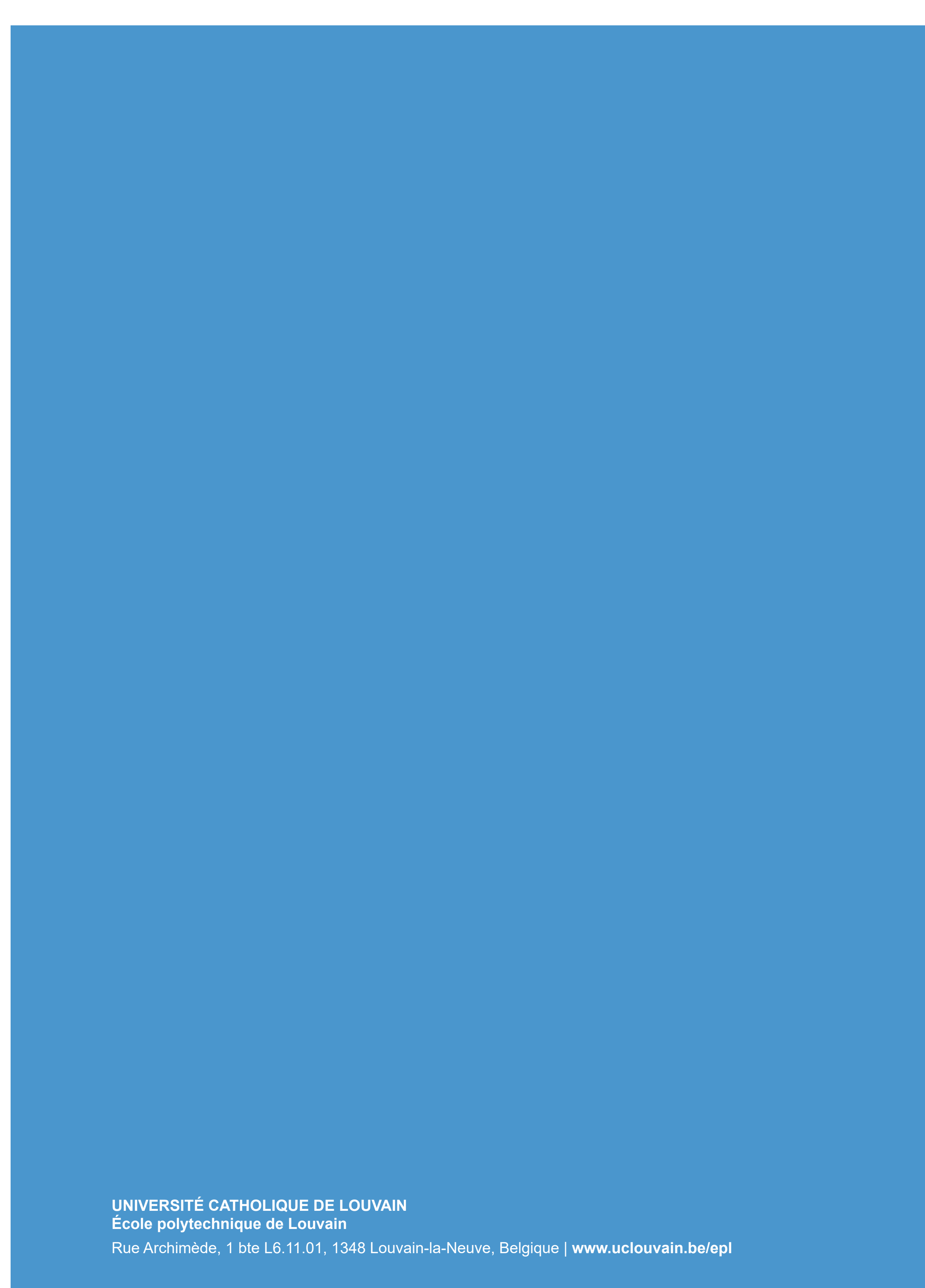
[25] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. “An experimental evaluation of selective mutation”. In: *Proceedings of 1993 15th international conference on software engineering*. IEEE. 1993, pp. 100–107.

[26] Mike Papadakis et al. “Mutation testing advances: an analysis and survey”. In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[27] Dennis Reuling et al. “Fault-based product-line testing: Effective sample generation based on feature-diagram mutation”. In: *Proceedings of the 19th International Conference on Software Product Lines*. 2015, pp. 131–140.

[28] Thomas Thüm et al. “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[29] Jie Zhang et al. “An empirical study on the scalability of selective mutation testing”. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE. 2014, pp. 277–287.



UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**