UCLouvain

epl

École polytechnique de Louvain

Testing the design of context-oriented software through mutation testing

Author: **Walter Merlini**
Supervisors: **Kim Mens, Pierre Martou**
Reader: **Charles Pecheur**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

In today's software development, building applications that can adapt their behavior according to different measurements gathered by the system has become more important. These information are called context and such applications are called context-aware applications. In parallel, programmers use more and more software product line methods like feature models. In recent years, a group of researcher at UCLouvain created a development method based on a model combining feature models and context models called Feature-Based Context-Oriented Programming. This master thesis starts from a previous work that defined theoretical basis to use mutation testing on this new type of models as well as a first implementation of a mutation testing tool based on a series of questions asked to the user. We will then see a new implementation of this tool designed to be more scalable and discuss a more efficient way to generate questions to the user.

Acknowledgment

I would like to thank first my two master thesis supervisors Prof. Kim Mens and PhD student Pierre Martou for their precious help throughout this academic year and the motivation they gave me during the writing of this master thesis. I would like to thanks Audric Deckers who gave me access to all his work on the subject from his master thesis.

I am also grateful to my family and my friends for all the support they gave me during the last 6 years at UCLouvain. A special thanks goes to the EPL-exchange staff at UCLouvain in Belgium and the international office center of the NAT-TECH faculty of Aarhus University in Denmark who gave me the amazing opportunity to spend the first semester of this academic year abroad.

Contents

1 Introduction **4**

2 Background **7**
2.1 Feature-based context-oriented programs . . . 7
2.1.1 Context-oriented programming . . . 7
2.1.2 Feature-based context-oriented programs . . . 9
2.2 Mutation testing . . . 12
2.2.1 Mutation testing in general . . . 12
2.2.2 Mutation testing applied to FBCOP . . . 14
2.3 Summary . . . 16

3 Case Study **17**
3.1 The context and feature model . . . 17
3.2 The context-feature mapping . . . 20
3.3 Summary . . . 23

4 Old mutation testing Tool **24**
4.1 Overview . . . 25
4.1.1 Workflow & architecture . . . 25
4.1.2 Detailed structure . . . 27
4.2 Analysis . . . 30
4.2.1 Error messages . . . 30
4.2.2 Bad smells . . . 31
4.2.3 Faults identification . . . 32
4.3 Summary . . . 33

5 Form and interpretation **35**
5.1 Preparing the survey . . . 35
5.2 Interpreting results . . . 36
5.2.1 Model files . . . 36
5.2.2 Mutation testing tool . . . 37

5.2.3 Results . . . 38
5.3 Summary . . . 39

6 New mutation testing tool **40**
6.1 File Parsing & Model creation . . . 41
6.1.1 Context-feature model . . . 41
6.1.2 Mapping model . . . 42
6.2 Sub-models . . . 43
6.3 Mutations . . . 45
6.4 Question generation . . . 47
6.5 Recommendations . . . 50
6.6 Summary . . . 51

7 Improving questions generation **52**
7.1 Theoretical foundations . . . 53
7.1.1 Using subsets for anticipations . . . 53
7.1.2 Using super sets for anticipations . . . 54
7.2 Questioning strategies . . . 54
7.2.1 Decision tree . . . 55
7.2.2 Decision tree with feedback . . . 56
7.3 Summary . . . 57

8 Validation **58**
8.1 Validating the re-implementation . . . 58
8.1.1 Methodology . . . 58
8.1.2 Analysis . . . 59
8.2 Threats to validity . . . 65
8.3 Future work . . . 65
8.3.1 Adding new mutants operators . . . 65
8.3.2 Cross-tree constraints . . . 65
8.3.3 Xml files parser . . . 66
8.3.4 OrToOpt pruning . . . 66
8.4 Summary . . . 67

9 Conclusion **68**

A Form results **70**

B Class diagram **78**

C New mutation implementation **80**

Chapter 1

Introduction

For some applications it can be useful to adapt their behavior according to certain contexts in which this application is running. For such applications an interesting approach could be to use the context-oriented programming (COP) paradigm. COP consists of creating a program that can adapt its behavior at runtime with the variation of some characteristics called contexts. Applications created through this approach are called context-oriented applications or context-aware applications. In order to facilitate the process of creating context aware applications, we need a development method allowing to easily create context-oriented programs. In 2022 an approach called Feature-Based context-Oriented Programming [1] (FBCOP) was introduced by B.Duhoux.

Imagine a messaging application running on a mobile phone. An FBCOP approach would be to first define a model representing different elements representing the context of the phone. These contexts could be a bad network connection, whether the localization is activated or not and many other things. On the other side we can also create a model using the same approach with the features of the program. What type of messages can be sent, are we using an alarm and what type of ringtone can be used are a few examples of features that could appear in this kind of application. Additionally to these two models, we would need a mapping model to express which feature(s) will be activated or not according to which context(s) are activated. This mapping model would express how the behavior of the program changes according to its state, it is simply modeled as a table indicating which context(s) activate which feature(s). When we put those three models together, we obtain a new model called a Context-Feature Mapping (CFM) model

To verify the correctness of such a model, we need to find erroneous configurations (set of activated context/features) among the space of all possible configurations. An erroneous configuration is a configuration that seems realistic

but does not respect certain constraint. In order to test efficiently a CFM model, we need an appropriate testing approach for this kind of model. For this, different solutions have already been tested such as visualization tools [2, 3, 4] or a Combinatorial Interaction Testing (CIT) tool [5].

Even thought these methods have helped the designer to verify his model's correctness, it did not really helped to know if the model respects correctly the design imagined for the program. This causes another problem, even if the models looks correct from a CIT point of view, it might not respect correctly each requirement made for the program. This can cause serious design errors leading to catastrophic consequences in some cases. For instance, if we take the case of any website using a database with restricted access according to the type of user. We could define two contexts, *adminUser* and *normalUser*. On the other side we could define two features that would be *fullAccess* and *restrictedAccess*. If the *normalUser* context is mapped to the *fullAcces* feature and the CIT tester does not point this problem, there is nothing that could prevent such a system to be implemented, but it is clear that the consequences would be catastrophic if we let every user have a total access to a database. In our example of a messaging application, a design error does not necessarily lead to disastrous events but might cause some behavioral changes that are not suitable for the user's experience.

Recently, a new approach has been used based on a technique called mutation testing. Mutation testing is a method used to verify and evaluate the ability of a unit tests suite to detect bugs in a function or a method. To do so, we purposely add small errors in the source code that we call mutations. This new approach consist in creating those small mutations in the model. Mutations coming out of the model will be used to generate different questions. Then, according the answer of the designer, the testing tool can detect potential errors in the model and generate recommendations to fix them. The first version of this tool was created by A. Deckers in his master thesis [6] and served as a starting point for this master thesis.

This master thesis will start by giving a theoretical point of view about FBCOP and mutation testing in chapter 2. In chapter 3 we will present a example of a CFM model that will be used as a guideline for the rest of the master thesis to have a more illustrative explanation of how the tool works. We will then present the tool created by A. Deckers in chapter 4, from now on this tool will be referenced as "the old tool". This chapter will also show different bad smells that we identified as well as some bugs that were found. We also modified the old tool by adding error messages that will help the user to identified syntax errors in his input by showing

the location of the error. In chapter 5, we created a feedback survey, this survey was answered by students who tested the old program. Their answers allowed us to have a global idea of how users experienced the tool and also find other bugs that we could not see before. From the result of the form, we decided to perform a re-implementation of the old tool described in chapter 6. This re-implementation was made to give more scalability to the program and corrected errors that we found previously. Chapter 7 will explain a new feature added in the testing tool. We added two new algorithm based on a decision tree to improve the way the program asks questions. Finally, chapter 8 will cover the validation of the new program by showing that previously presented bugs are fixed and also propose potential works that can be conducted as a continuation of this master thesis.

Chapter 2

Background

This chapter is here in order to understand correctly two key aspects of this master thesis. The first section will explain what is a feature-based context-oriented program (FBCOP). The second part will then explain the concept of mutation testing. We will explain how this method is usually used in a Software Quality Assurance activity and then we will see how it can be used in FBCOP.

2.1 Feature-based context-oriented programs

As explained in the introduction, the goal of FBCOP is to allow a program to adapt its behavior at runtime in order to adapt the program to the context in which it evolves. This section will explain in detail what are the differences between context-oriented programming (COP) and feature-based context oriented programming. In a few words, COP is a programming paradigm and FBCOP is a variant of COP inspired by feature modeling.

2.1.1 Context-oriented programming

When we think about a hardware or a software system, usually we express its use in term of input and output. Inputs are generally given by a human so the system can process it and give an output. Although this vision is very simple and can still be used in many systems, it became too restrictive. Today's systems must be able to adapt to data that are not provided by the user but can describe the environment in order to better serve its user [7]. These systems are also called context aware because they continuously sense their context. A context can express many things. It can be a measure of environmental properties like the humidity, the noise or the natural light but it can also be an expression of the internal state of the system like the battery, the time, the CPU temperature,... . Actually, everything can be used

as a context as long as it is not an input given by the user and it can be measured.

To implement a context-aware program, there are many different methods. The first method uses conditional statements. For each function to implement, we use multiple conditional statements to find which possible context matches with the current one and adapt the function behavior. The following piece of code illustrates how a method used to notify a phone user of an incoming call would look like [8].

Algorithm 1 Use of conditional statement for a context-aware function.

if phoneIsOffHook() **then** $playWaitingSignal()$
else if quietEnvironment() **then** $playVibratingRingtone()$
else playRingtone()
end if

In this example, the context defines which type of notification will be applied. We used 3 possible contexts, the phone can be off the hook, in a quiet environment or in none of the two. The last context defined in the code is always the default context. When one of the conditions returns `True`, that means that we found in which context the phone is working and we can use the appropriate reaction of the program to the incoming call. The main advantage of this method is its adaptability. If we want to add a context to the function we simply have to add another if statement. On the other side, this method implies that we have to check every possible context every time we call a function that is adaptable.

The second option is to use a special software architecture [9]. This method requires the use of a design pattern adapted to the problem. The example below uses a Strategy design pattern this design pattern allows the program to define different algorithms for the same problem and choose at runtime which one will be applied [10]. Therefore, we can really apply this to a COP example making the program more modular by defining a strategy for each context.

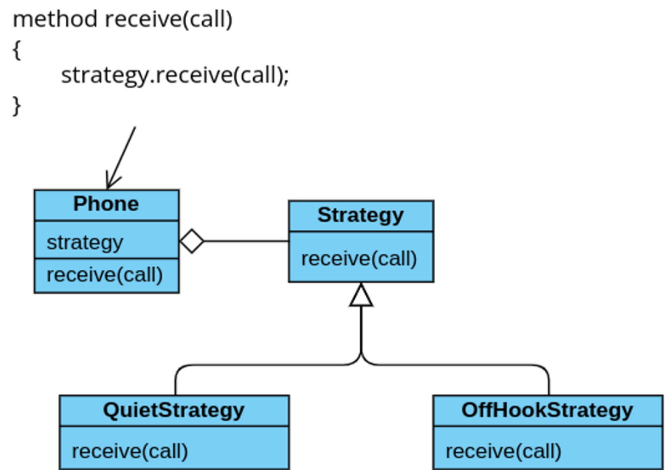


Figure 2.1: Strategy design pattern applied to a Context Oriented Program.

In this solution we have a central class called **Phone** that represent our system. In this system we can define a *strategy* variable that contains an instance of a subclass of **Strategy** containing a method called *receive*. According to the current context, *strategy* will contain an object containing the *receive* method. This object will then use its own implementation to process the call.

2.1.2 Feature-based context-oriented programs

In order to create a context-oriented program, we have to find a way to model these programs. In 2022, Benoît Duhoux introduced in his PhD. thesis an approach called Feature-Based Context-Oriented Software Development [1]. This approach intent to create a model that allows us to better understand how the program will behave according to its state (context). This model is really important as it allows the developing teams to have a better understanding of what the program should do.

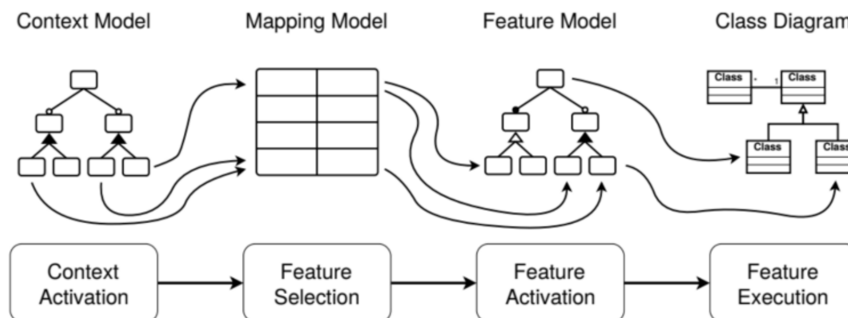


Figure 2.2: Control flow of the FBCOP methodology [1].

Feature-oriented domain analysis

The FBCOP method uses a type of model called feature model. A feature model is a concept used in software product lines to distinguish common features and features that may not be common in a family of systems. This kind of model was introduced by Kang et al. in 1990 [11] to define Feature-Oriented Domain Analysis (FODA). For a more visual explanation let's consider the following example of an e-shop application:

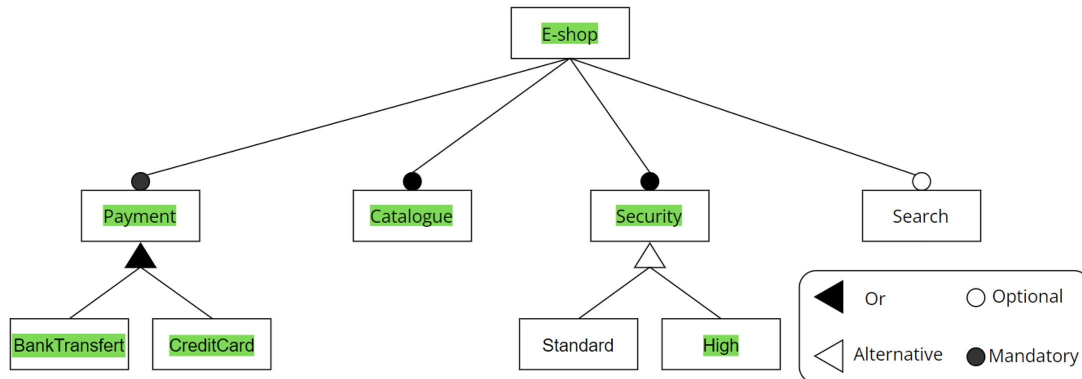


Figure 2.3: Feature model of an E-shop application [8].

This hierarchical model shows different features that can be used in a e-shop application (like amazon, or e-bay). We can find different commonalities like a payment method, a product catalog or a security level but also a variability called *search* that represents a feature allowing the user to search for a product. The root of the model is the name of the application and we also notice different signs put on each feature. These signs represent constraints. A constraint is usually a boolean operator that is here to keep some consistency in the model, they also limit the number of valid configurations of the model. In this example we find 4 different types of constraints.

- The *Mandatory* constraint : This constraint states that the concerned feature **must** always be selected if its parent feature is selected. This constraint defines the commonalities in the domain. Note that an *And* constraint is equivalent to a feature with a mandatory constraint on each child.

- The *Optional* constraint : This constraint states that the concerned feature **can** be selected if its parent feature is selected. This constraint defines variabilities in the domain.

- The *Or* constraint : This constraint states that **at least** one sub-feature of a given feature must be selected in the configuration.

- The *Alternative* constraint : This constraint states that **exactly** one sub-feature of a given feature must be selected in the configuration.

A configuration of a feature model is a subset of all features that are present in the model. Features that are present in this subset are called "active" features and the others are called "inactive" features. When a configuration respects all the constraints of the model, this configuration is said to be valid. In our case, a configuration with *E-Shop*, *Payment*, *BankTransfert*, *CreditCard*, *Catalogue*, *Security* and *High* as shown in Figure 2.3 is a valid configuration but other valid configurations exist. If we remove *Catalogue*, the configuration becomes invalid as we do not respect a *Mandatory* constraint. When the model does no contain any valid configuration, it is said to be inconsistent.

By using a feature model, we can define easily many different programs from the same model by defining multiple valid configurations and thus use a software product line more efficiently.

Context-feature mapping model

Using the same kind of model used in section 2.1.2 we can use an approach defined by Desmet et al. in 2007 [12]. Their paper proposed to use the same semantics of FODA applied to contexts instead of features. This way we are able to build a context model defining all the contexts that can be used by an application.

To build a context model, we put in relations every possible contexts in which the application could be used. Like a feature model, we will have contexts and sub-contexts that are related through constraints. Constraint used in a context model are the same as those used in a feature model.

Now that we have our context model and our feature model, we have to find a way to know when a feature can be activated or not during the use of the program. The activation or deactivation of a feature has to depend on the current context of the system. Therefore we have to find a way to associate activated contexts with the features that they activate. For this we use a third component : the mapping model. The mapping model is placed between the context model and the feature model. It simply maps a context or a group of contexts with a feature or a group of features to indicate which feature becomes activate for each context. The image below represents a simplified version of a context-feature mapping model that will be introduced in chapter 3.

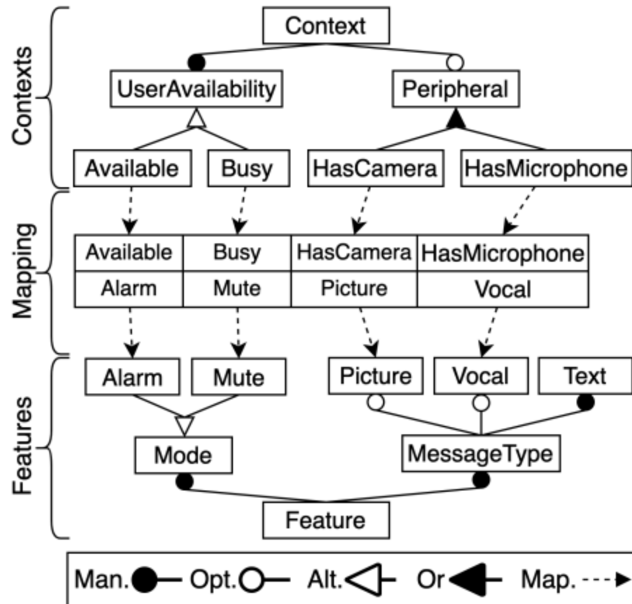


Figure 2.4: Example of a CFM model [6].

Once our model is correctly defined, the next step is to verify it in order to avoid design flaws tha would create errors in the future program.

2.2 Mutation testing

This section explains the other crucial part of this master thesis's subject, mutation testing (MT). Defined first in the 70's [13], mutation testing uses modified versions of a program's code called mutants in order to know how good a test suite is to find bugs. In order to have a good idea of how this method will be applied to FBCOP systems, we will first dive into a more detailed explanation of MT. Once this is done we will see how the classical method will be slightly modified to test a CFM model.

2.2.1 Mutation testing in general

Mutation testing was initially introduced in 1978 by DeMillo et al. [13] in order to help program testers to define test suites. The concept of MT is to create many different versions of the same method or function that are called mutants [13]. To generate a mutant, we slightly modify the original code in a different way for each mutant. The goal here is to create multiple implementations of a same function and see if a test suite is able to find the bug. If the bug is detected (i.e. at least

one test crashes) we say that the test suite killed the mutant. To be efficient, a test suite must kill as many mutants as possible. The following example shows a piece of code that has to be tested and some mutants generated from the code.

```
#Original code
def sumOfSquare(n):
    s = 0
    for i in range(n):
        s+= i * i

    return s
```

```
#Mutant 1
def sumOfSquare(n):
    s = 0
    for i in range(n):
        s+= i + i

    return s
```

```
#Mutant 2
def sumOfSquare(n):
    s = 1
    for i in range(n):
        s+= i * i

    return s
```

In order to work correctly, MT uses two important hypothesis that allows him to work correctly. The first hypothesis is called the competent programmer hypothesis [13]. This hypothesis states that competent programmer can make mistakes in their code, but in the end their code will be close to correct and will only need a few modifications. For instance, if a programmer is asked to write a function that sorts an array of integers in increasing order, some mistakes like a wrong comparison operator or an algorithm that actually sorts integers in decreasing order are possible but the behavior of this algorithm is close to the desired algorithm. Usually the algorithm will not do something unrelated like calculating the sum of each integer in the list. The second hypothesis is "the coupling effect" [13]. The coupling effect is present when major errors cannot be found because other minor errors are not detected. Therefore, if a test suite is able to find small errors in a code, then it will be able to detect more important ones. This is why it is important to assure that a test suite is able to detect small mistakes.

Another way to see MT is to define it as a white-box coverage testing method [14] to test a test suite. As we said previously, we want to be sure that our test suite is robust enough to find bugs. When we start a test suite on a set of mutants, we can see how many mutants survived and see what kind of bugs are not taken into account in our test suite. We can also define some metrics in our mutants set. From a number k of killed mutants and a number s of survivors, we measure the coverage of the test suite by $\frac{k}{k+s}$. A good coverage would then kill as many mutants as possible. We can also estimate how many faults are left to be discovered in the program. If we assume that the proportion of killed mutant is the same as the proportion of faults that are discovered, we can estimate the total number of faults N with the following equation : $N = \frac{S*n}{s}$ where S is the total number of mutants, s the number of killed mutants and n the number of faults already discovered [15].

2.2.2 Mutation testing applied to FBCOP

To apply MT on CFM models we have to find a new way to apply what was defined in the previous section. In CFM models, mutation testing is not used in the traditional way. Instead of using it as a way to test the robustness of a test suite. We will define mutations to directly test the model. The goal here, is to create a tool able to generate mutations directly in the model. Those mutations will be used to generate questions for the model designer to answer. Answers to those questions will later allow the tool to know whether a fault is present or not in the model.

This method is inspired by the work of Arcaini et al. [16] in 2015. Their article explain how mutations applied directly on a feature model can be used to test the model. Their operating mode was to first translate the model into a boolean proposition. The mutant generation step consist in applying small changes to the boolean formula. Although this seems easy to create, the mutant generation tool has to take into account one crucial aspect, We want to be sure that the mutant is not equivalent to the initial model.

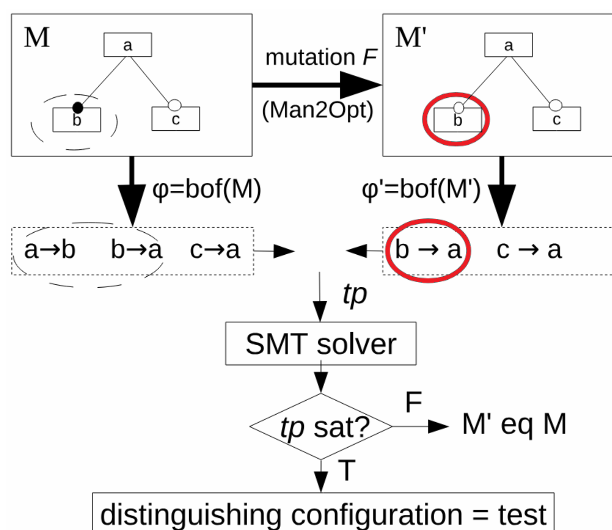


Figure 2.5: Algorithm proposed to identify distinguishing configurations [16].

To address this problem, Arcaini et al. [16] decided to search for distinguishing configurations. A distinguishing configuration is a configuration that is valid in the original model but not in the mutant or vice-versa. The process is the following: we transform the model and its mutant into two boolean formulas $\mathcal{M}$ and $\mathcal{M}'$. The

goal will then be to find an interpretation of those formulas is valid for only one of them. By using a **XOR** operator between $\mathcal{M}$ and $\mathcal{M}'$, we define a new formula that is satisfied only when $\mathcal{M}$ is satisfied but not $\mathcal{M}'$ and vice-versa. The obtained formulas now have to be inserted into a SAT solver in order to know whether there is a distinguishing configuration or not.

Regarding which mutation we are going to use, Arcaini et al. gave us different examples of possible mutations that can be applied to a feature model [16, 17] . Figure 2.6 gives us some examples of what kind of mutations can also be applied in a CFM model.

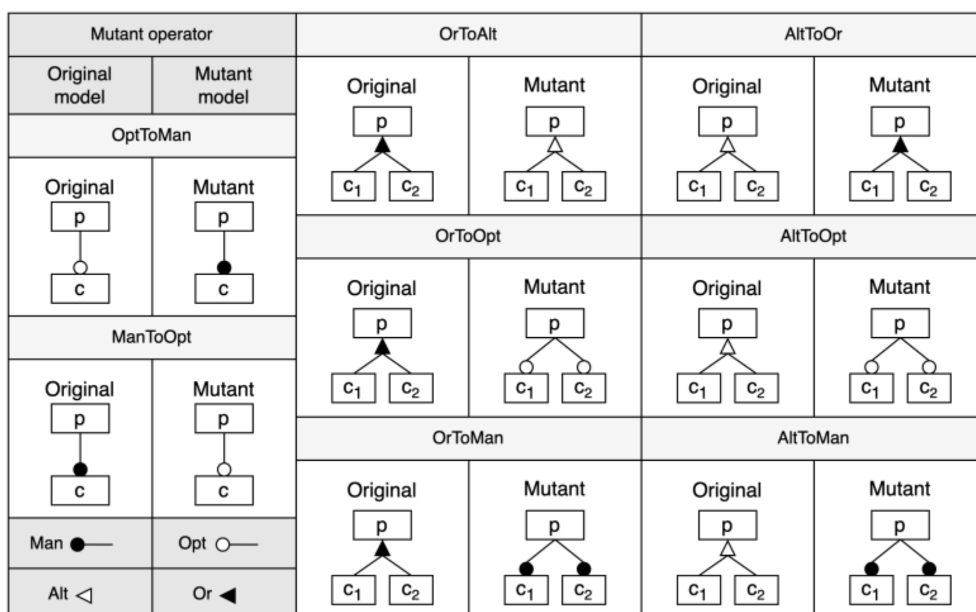


Figure 2.6: List of possible mutations on a CFM model [6].

In this master thesis we will only use a few type of mutations that are presented in figure 2.6. We will use **OrToAlt** to transform *Or* constraint to *Aternative* constraint, **AltToOr** to apply the opposite mutation, **OrToOpt** to go from a *Or* constraint to a set of *Optional* nodes, and **ManToOpt** to allow *Mandatory* node to become *Optional.* Those mutations are chosen to respect a certain rule that will be described in more detail in chapters 6. The idea here is to use the competent programmer hypothesis and use mutations to transform a constraint into a new one with a similar, though distinct, behavior.

2.3 Summary

In this chapter we introduced two key theoretical concepts for this master thesis. The first one is the Feature-Based Context-Oriented programming method or FBCOP. This method is used to build context-oriented applications thanks to a process method called feature-oriented domain analysis (FODA) where we define a model for a family of systems showing which features can be relevant to implement. Then, by applying FODA mutants to a feature model but as well to create a context model, and defining mapping rules between those two models, we are able to create a CFM model that will tell us which feature can be activated depending on which context is activated.

The second part of this chapter concerned mutation testing. MT was invented in 1978 in order do allow software testers to define more robust test suite by defining multiple different versions of the same programs that purposely contain some faults. With these faults we are able to define how good a test suites is to cover all possible faults that a program can have.

Lastly, we applied MT to CFM models by using a method inspired by Arcaini et al. [16] that initially tested feature models only. The goal now is not to test a test suite but to directly test the model by defining small mutations of the model. Those questions would then generate a set of questions whose answer will help us to find potential faults in the model.

Chapter 3

Case Study

In this chapter we will introduce a running example of an application that can be modeled with a CFM model. The purpose of this example is to have a better understanding of how the mutation testing tool works. The example used here, is a smart messaging application. This is a good example because it is really easy to guess how this kind of application can be done in the context of FBCOP as messaging applications are by most of smartphone users on a daily basis.

This chapter will first go through the presentation of the context model and the feature model. Then we will see how these two models are linked with a mapping model. It is important to note that some errors are purposely introduced in the model in order to validate in chapter 8 the new version of the tool that will be presented in chapter 6.

3.1 The context and feature model

A context model is used to represent and link different contexts a system can live through. A context can be any internal or external information usable by the system to define a proper reaction [1]. As an example we can activate a context called *LowBattery* when the battery level drops below 15% or we can also define a *Night* context that becomes active after 10 p.m. In our example only a portion of all possible contexts will be defined in order to keep the model simple. Of course, in a real life model we would have many more contexts.

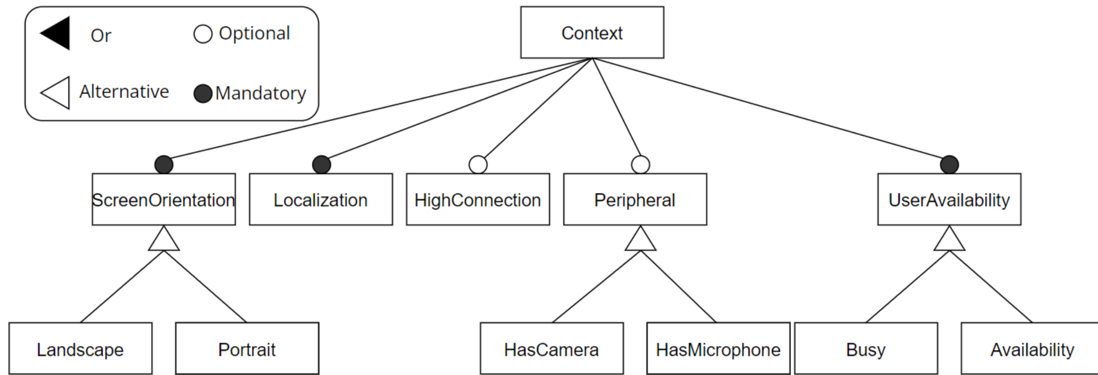


Figure 3.1: Context model used for our example application.

As every context model should, it is shaped like a tree and the root node is simply called "Context". The reason why it is built like a tree is because we can define different groups of contexts (or parent contexts) of higher level of the tree and from those groups we can also define subgroups and/or contexts on a lower level. In the context model above, we can identify different (groups of) contexts:

- *ScreenOrientation* : This group defines two different orientations a smartphone can have : *Landscape* and *Portrait* .
- *Localization* : This context describes the geographical position of the user.
- *HighConnection* : This context becomes active when the user has a good connection. To ease the comprehension we usually make abstraction of the metric used to (de)-activate the context.
- *Peripheral* : This parent context regroups the possibility for the device to be equipped of a camera (*HasCamera*) and a microphone (*HasMicrophone*).
- *UserAvailability* : This part simply indicates whether the user is *Busy* or *Available*

The feature model can be interpreted in the same way as the contexts model. We have the tree shaped model with features instead of contexts and the root node is called "Feature". Instead of directly define features we can regroup them using parent features the same way as we defined parent contexts.

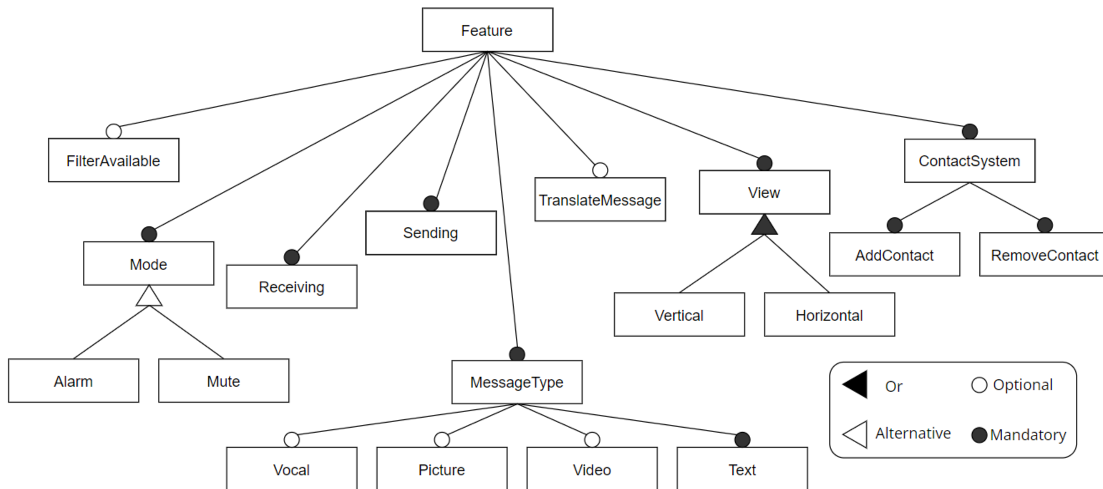


Figure 3.2: Feature model used for our example application.

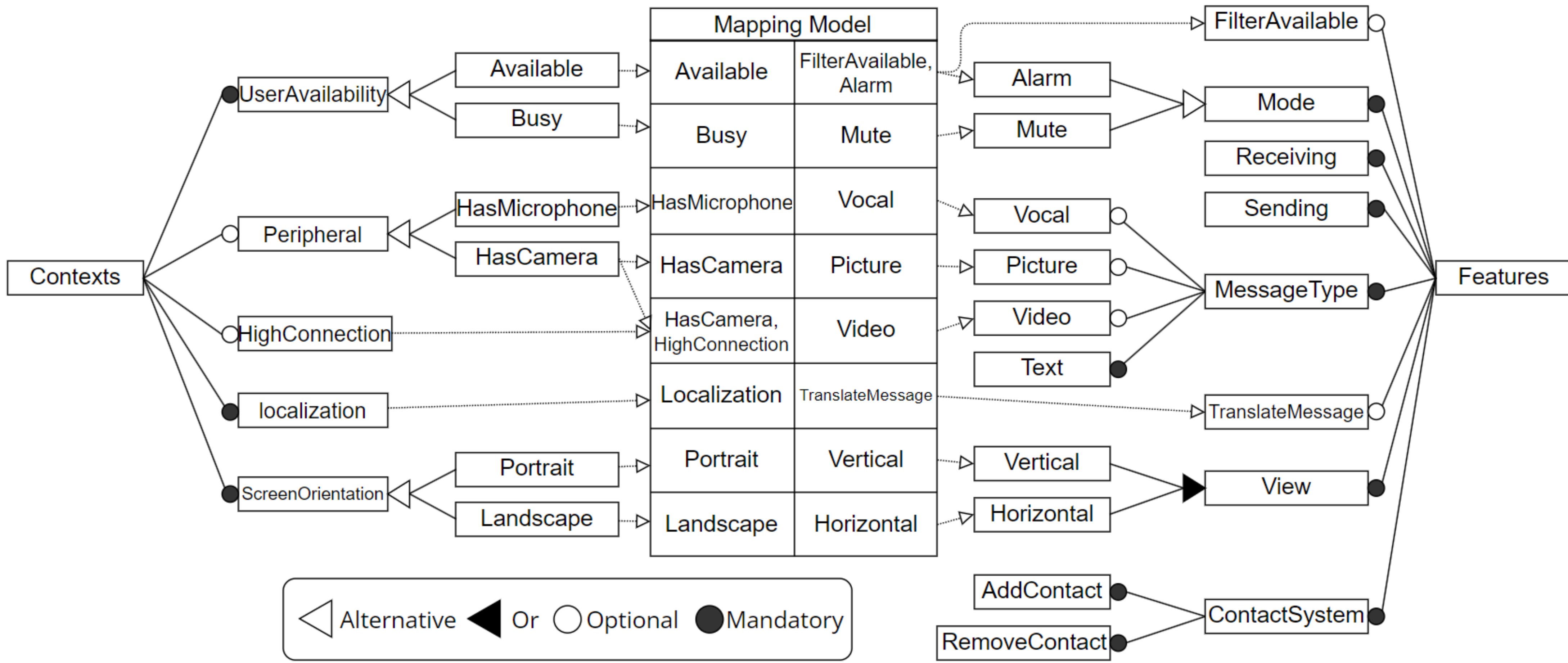
We identified the following list of features :

- *FilterAvailable* : This features allows a user to search for other available users in order to initiate a conversation with them.
- *Mode* : Here it is about the sound that can be produced from the app when the user receives a notification. The *Mode* can either be set on *Alarm* and produce a sound when a message is received for instance or it can be set on *Mute* when the user does not want any sound coming from the app.
- *Receiving* : This just refers to the ability to receive messages on the app.
- *Sending* : Allows the user to send messages.
- *MessageType* : Defines the kind of messages that can be sent by the user. Here we have, *Pictures*, *Vocal* messages, *Videos* and simple *Messages*.
- *TranslateMessage* : When this feature is enabled, the application automatically translate messages received by the user.
- *View* : Decide how the view of the application can be and how each element present on the screen can be placed. The view can either be *Horizontal* or *Vertical*.
- *ContactSystem* : It is a common feature to many messaging applications. We leave the possibility to the user to *AddContact* or *DeleteContact*.

Something we notice on those models is that different signs are present on each node. These signs are called constraints and they can be of different type. A constraint is here to put a condition on the activation of one or more nodes. They can be seen as a boolean proposition where a node is a variable. These boolean propositions are made in a way that they must be `True` at any time. When we say that a proposition must always be `True`, we do not mean that it has to be a tautology (`True` in all interpretation) but it must always be in an interpretation resulting to `True`. For example, in the context tree *Landscape* and *Portrait* contexts are in an alternative constraint. That means that either *Landscape* can be activated (i.e. `True`) or *Portrait* but never both at the same time and always one of them must be activated. This can be interpreted in the following manner : *"The screen orientation can either be in landscape or in portrait."*. The black dot at the top of *SceenOrientation* means that this context is mandatory and therefore must always be activated. On the contrary, we can also define non mandatory nodes like *peripheral* meaning that this context can be deactivated.

3.2 The context-feature mapping

Once we define which contexts can affect our application and what features can define our application, we now have to map those two models together. Mapping those contexts means that we have to decides which context(s) when (de)activated will results in the (de)activation of one or more feature. This is done using a two columns table put between the two models define earlier.



(a) Demonstration of how a context model is mapped to a feature model

For each row, the table can be interpreted this way : "When every context on the left side are active, then every feature on the right side are active as well". This way of reading really puts in harmony how to interpret the whole model and gives a way better understanding of how the application should work. Let's see in our example, how each mapping element can be interpreted :

- When the user is *Available*, then the user is able to search for other available users through the *FilterAvailable* feature and the *Alarm* mode will be turned on.
- When the user is *Busy*, then the *Mute* mode will be turned on.
- If the device used by the user has a *Microphone*, then the user is able to send *Vocal* Messages.
- If the device used by the used has a *Camera*, then the user is able to send *Pictures*.
- If the device used by the user has a *Camera* and the user has a *HighConnection*, then the user is able to send *Video* messages.
- If the device used by the user allows the access to the *Localization* of the device, then the app enables the *TranslateMessage* feature.
- If the screen is *portrait* oriented, then the *View* will be *Vertical*.
- If the screen is *landscape* oriented, then the *View* will be *Horizontal*.

From this example we can derive one property of this mapping model : when a context or a group of contexts is mapped to (= activates) a group of features, it is equivalent to say that this context or this group of contexts is mapped to each feature of the group individually. If we take a look at the first row, we have a mapping from *Available* to *FilterAvailable* and *Alarm* but we could also write the same thing with two mapping, *Available* to *FilterAvailable* and *Available* to *Alarm*. On the contrary, the opposite property is not correct. We cannot dissociate all the contexts from a group of contexts to create more mappings. On the fifth row in the the model above, *HasCamera* alone or *HighConnection* alone are not enough to activate *Video* messages, both of them need to be on to activate the corresponding feature. These two properties will be very helpful later to create sub-models.

3.3 Summary

This chapter was used to cover the example of a messaging application for which we defined a CFM model. To do so we built a feature model as well as a context model and a mapping model between them. The goal was to have a consistent example through the rest of this master thesis to better understand how the testing tool would work on it. For a future validation of the messaging application model, we inserted some errors on purpose and we will see if the tool is able to catch them or not.

Chapter 4

Old mutation testing Tool

Before analyzing the latest version of the testing tool created for this master thesis. This chapter will present and analyze the old version. The old version was implemented by A. Deckers in his master thesis [6]. We will start by an overview of the program by describing first the workflow of this release which is basically the same in the new version presented in chapter 6. To finish the overview we will give a more detailed explanation of the code. The second part of this chapter will present 3 contributions added to the program for the purpose of this master thesis. The first one will be the addition of error messages, the second contributions regards the identifications of bad smells that are present in the code base and for the last contribution we identified some faults in the program that will be fixed in the next version presented in chapter 6.

4.1 Overview

4.1.1 Workflow & architecture

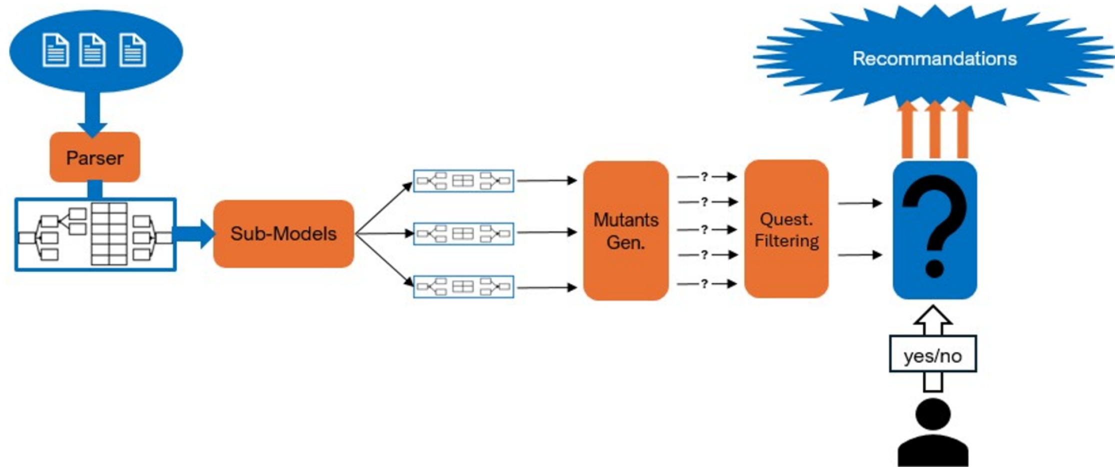


Figure 4.1: Workflow of the mutation testing tool.

The workflow of the program starts with the user input. In order for the program to understand correctly the input model, P. Martou defined a syntax that can be used to encode CFM models [5] by using 3 *.txt* files. The two first files represent the feature and the context model and have the same structure, they are respectively called *features.txt* and *contexts.txt*. On the other hand, the third file represents the mapping component and is simply called *mapping.txt*

The feature model as well as the context model have a similar appearance when it comes to describe them. They are both trees containing nodes and constraints. In order to allow the user to define them, we need to define a syntax allowing a clear description of the model. The syntax used for this program works as following. Each constraint is defined line by line and contains 3 parts separated by a "/" character. The first part is the name of the parent node of the constraint. In the second part, the user must simply put the name of the constraint and the third part gives the name of children nodes present in the constraint. It is important to note that if we want to write a node as a parent node for a constraint, the node must have been defined previously as a child node. The root of the tree which cannot be defined as a child node anywhere in the model must be defined on the first line of the model. The name of the root must either be *Feature* or *Context*. You can find below how the context model and the feature model defined in Chapter 3 would be encoded with this syntax.

contexts.txt

```
Context/Mandatory/UserAvailability-Localization-
 ↪  ScreenOrientation
Context/Optional/Peripheral-HighConnection
UserAvailability/Alternative/Available-Busy
Peripheral/Alternative/HasMicrophone-HasCamera
ScreenOrientation/Alternative/Portrait-Landscape
```

features.txt

```
Feature/Mandatory/Mode-Receiving-Sending-MessageType-View-
 ↪  ContactSystem
Feature/Optional/FilterAvailable-TranslateMessage
Mode/Alternative/Alarm-Mute
MessageType/Mandatory/Text
MessageType/Optional/Vocal-Picture-Video
View/Or/Vertical-Horizontal
ContactSystem/Mandatory/AddContact-RemoveContact
```

The mapping has his own syntax, each line of the file represents a mapping between a group of contexts and a group of features indicating which features are activated when the whole group of contexts is activated. To represent this, a line is composed in the following way. Each context are separated by a "-" in order to differentiate them. Then we have the keyword *-ACTIVATES-* followed by each features also separated by a "-". The code for the mapping model defined in the previous chapter is presented below.

mapping.txt

```
Available-ACTIVATES-Alarm,FilterAvailable
Busy-ACTIVATES-Mute
HasMicrophone-ACTIVATES-Vocal
HasCamera-ACTIVATES-Picture
HasCamera,HighConnection-ACTIVATES-Video
Localization-ACTIVATES-TranslateMessage
Portrait-ACTIVATES-Vertical
Landscape-ACTIVATES-Horizontal
```

These files that we just described will then be used in the testing tool. First of all, *.txt* files will be parsed to check if they have a correct syntax and if their composition is coherent (i.e. a node present in the mapping is always defined in a context/feature file.). If the parsing works correctly, the tool is then able to create a representation of the whole CFM model. This model, will then be examined to extract connected pair. A detailed definition of Connected pair will be explained in chapter 6 but for now, we define a connected pair as a interconnection of a context and a feature where at least one child of the context activates at least one child of the feature [6].

Connected pairs are very essentials because they will be the starting point of the mutation process. In A. Deckers's tool [6], mutations are defined and performed under certain conditions. Each connected pair generated will then be checked to see if some mutations can be applied. If so, a mutation is created by simply switching a constraint in the connected pair. With those mutations stored, each mutation applied to a connected pair will result in the creation of questions asked to the tool user. The very last step is to define some recommendations that will be generated according to the user's answers. After replying to each question, the tool displays a mutation score corresponding to the rate of constraint that are correctly defined in the model.

4.1.2 Detailed structure

The source code of the program is divided in three different files, the fist file is called *launcher*. It works simply by using the name of each input files and use them to create the CFM model with its mutations and questions. Once these questions are generated, the model created will start asking questions from mutations. Those questions will then be answered by the user and finally, recommendations will be generated from the user's answers.

The second file contains a class called **CFMmodel** and is the more important. As its name suggest, its goal is to create a CFM model but it is also responsible to parse input files, generate connected pairs and create mutations from which the module is going to generate a set of questions to ask. But before diving into this section, we need to understand properly how the third modules works. The third class is called **CFMnode**. The purpose of this module is to define a node in a context model or a feature model. The **CFMNode** class is only used to contain information about a node. This allows to keep the name of the node, its parent and its children. It is also possible to know in which connected pair a node is involved and what constraint contains the node as well as the type of tree (context or feature) to which the node belongs.

The first step of the second module is now to set up different instance variables or attributes that are used during the process. Those attributes will allow to first differentiate nodes from the context tree and nodes from the features tree. We will also keep in memory every connected pair generated as well as all the mutants and questions. To represent the mapping model, the **CFMmodel** class contains two dictionaries. The first one maps each contexts to every features it activates and the second one does the contrary by mapping each feature to every contexts that activates it. Once those attributes are set up, the next step will be to parse each input file.

As said previously in section 4.1.1, there are two different types of files regarding the syntax of the input. The first syntax to parse is the context and feature syntax that allows to create the context and the feature model. To parse this model we start by reading the file line by line using a line counter. The line counter is used when there is an error to raise. This way the user will know what is the problem with his encoding and form which line does the error comes from. It is also important to see that the file searched for the model actually exists and that it does not contain any empty line. Once these conditions are checked, we divide each line in different parts to isolate the parent node, the constraint and the children nodes. Of course if this format is not respected, an error is raised to the user. Once the division has been made, we can easily create a node for the parent node as well as every child node with the constraint written in the line. Each node are added to a common set containing. When a node is created, it is then added to the tree structure.

For the mapping model, the file is also inspected line by line and there also is a line counter to have information about encoding errors like in the context/feature model. In this case as we know that features and contexts are separated by an -*ACTIVATES*- keyword, we only divide the line in two elements, the context list and the features list. Then for each one of theses elements, we have to check that each feature and each context were already created in one of the two previous model files and that a node corresponding to this name already exists. The mapping element will be added in two dictionaries, one mapping contexts to features and one mapping features to contexts.

The next step in the workflow is the creation of connected pairs. For A.Deckers [6], a connected pair is created when one or more children of a certain context activates one or more children a feature node. The tool simply go through every context, every features and theirs children nodes. For every context-feature combination,

if there is at least one child of the context that maps to a child of the feature, a connected pair is created. The connected pair consist in a dictionary containing information such as context/features parents, node and constraint characterizing the connected pair.

The mutation step works pretty simply, we iterate through every previously created connected in order to find a mutation to apply. Mutations are applied according to the constraint of each connected pair. To decide which mutation to apply, we have to take into account two aspects. The first is the prioritization of the mutation. As explained in section 2.2.2, mutation testing uses the competent programmer hypothesis. From this assumption, we can state that a mutation should follow the following logic defined by A. Deckers [6]. This method is made in order to either relax a constraint to make it a bit less restrictive or to strengthen it by replacing the constraint by augmenting its restriction.

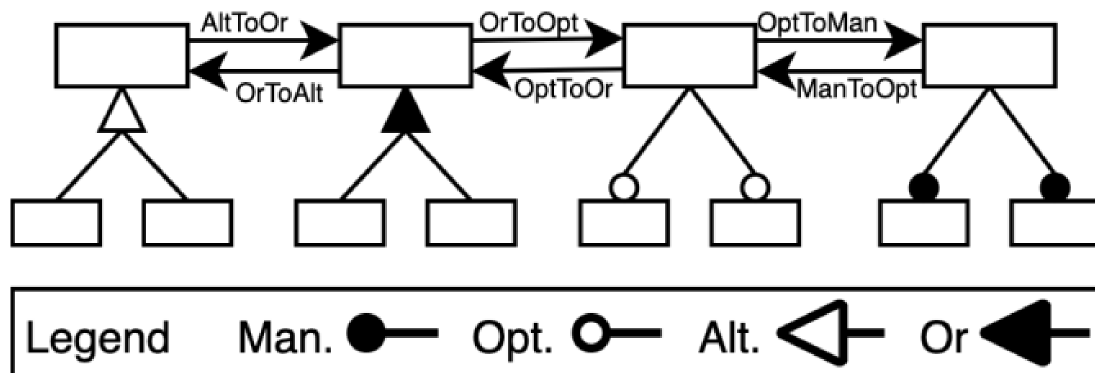


Figure 4.2: Constraint hierarchy for mutations.

To finish the workflow of this program, the last step consist in generate questions and recommendations if the answer to the question implies the presence of an error in the model. Each questions are created at the mutation step. That means that for each mutation created, a question is generated. Moreover each questions consists in knowing whether the mutation gets us to a correct configuration of the connected pair. When the user answers to a question, if the answer indicates a fault in the model, the tool will suggest to the user some modification that he could apply to his model. In the end, a mutation score is given to the user. This score express which proportion of questions did not helped to find errors so the higher the score is, the better the model is.

4.2 Analysis

4.2.1 Error messages

In chapter 5 will talking about a survey that was given to student of the course Software Maintenance & Evolution. This survey was made in order to have a feedback about their experience using the old version of the testing tool. In order to have a better experience and make sure that these students are able to use the program correctly, we made sure that they could have a clear error message in case there is an error occurring while parsing input files. These error messages were made to signal a user if there was a syntax error in an input file. To indicate which line was wrong, we used a line counter variable that is incremented by 1 for each line read by the parsing method. At first the tool was able to detect 3 different errors in the context and the feature files :

- The input file does not exist or it was not named correctly (*contexts.txt* or *features.txt*)
- The read line does not respected the required syntax (see 4.1.1)
- The name of the constraint given in the read line is unknown

Unfortunately, these errors were not enough, we found out that in some cases, the file was parsed but caused errors later in the workflow because the file was not written correctly. To correct that, we modified the parsing method to be able to detect the following errors:

- The same node name is found in both files (a context cannot be a feature and a feature cannot be a context at the same time).
- The parent node name in the read line does not exist.
- The root of the model has to be named *Context* or *Feature* according the file that is being read.

The tool also handles 3 different errors in the mapping file parser. These errors were already detected by A. Deckers but we decided to change the error message to help the user identify the location of the error

- The read line does not respected the required syntax.
- The mapping keyword -*ACTIVATES*- is missing or is not written correctly.
- The read line uses a context or a feature that was not defined in their respective file.

4.2.2 Bad smells

This section is aimed to have better understanding of which elements need to be changed in this code in the future. We will see that even if the code seemed to work at first, chapter 5 will show that this is not always the case. Sometimes, even if the program seems to work, when we test it with other inputs it can crash. This unfortunate event can be avoided first by having a bigger test suite, this will reduce the probability of having undiscovered errors. The second solution is to avoid using bad smells in the code. Bad smells are a set of bad practices that can make your code harder to maintain [18]. Bad smells also facilitate the introduction of bugs in our program code and when a bug is inserted in a bad looking code it is often really hard to find and then to fix.

Long method

Sometimes, a method is required to do multiple things to get to the desired output. Because of that, some programmers tend to create long methods with many lines of code in order to accomplish what the method is supposed to do. In this case, we have many methods of the **CFModel** class that contains way too many lines :

- **ProcessCFFile** - 81 line
- **ProcessMappingFile** - 43 lines
- **generateConnectedPairs** - 87 lines
- **generateMutants** - 129 lines
- **launchRecommendationSystem** - 42 lines

To solve this issue, the programmer should try to find a way to reduce the number of lines in the code to around 10 lines . Usually, when a method is very long, it contains several code repetitions that must absolutely be eradicated. Code repetition being sometimes considered as the worst bad smell [19]. A way to avoid this problem is to use a method extraction [20]. Method extraction consists in removing a piece of code from a method to create another method from it and replace this code by a call to the new function. It is very useful when we want to remove code duplication but it is also very powerful when it is used with a good naming of the method because we can then use it to separate each steps of a method by putting them in a different method for each step.

Large Class

The Large class bad smell occurs in different cases. First, when a class is really long compared to other classes of the program. Second, when a class contains too many attributes and third, when a class contains too many methods. The first version of the tool contains all 3 of those conditions. The old version contains only two classes, **CFMmodel** contains 13 methods (constructor excluded) and 539 lines while **CFMnode** contains only 2 methods and 31 lines.

There are many ways to reduce the size of a class. In the same logic as extract method, we would imagine to extract a class from the current class. This technique is used when classes like **CFMmodel** have more than one role. In Object-oriented programming, this is called the single responsibility principle [21]. This rule states that each class must have maximum one role in the program. This practice ensure that the class is not too long but also more readable and more maintainable [21]. This is also a good way to have a more modular program instead of putting the entire workflow into a single class

4.2.3 Faults identification

Besides errors that were not checked before, we also found faults for models that were encoded correctly. In the form that students filled in chapter 5 we asked them to give us their encoded models in order to test them and see why these models fail. For some models the output seemed correct at first glance, but for others we could notice very annoying phenomenon. In the image below, we isolated all the questions that were asked to the group from the output in order only show what is important for now.

```
1   -Do NotMessage context(s) have to be activated in any configuration? (yes/no):
2   -Is it possible for Work,Personal contexts to be activated simultaneously? (yes/no):
3   -Is it possible for Work,Personal contexts to be deactivated simultaneously? (yes/no):
4   -Is it possible for Work,Personal contexts and for TextToSpeech,PredefinedMessages,Messages
5       features to be activated simultaneously? (yes/no):
6   -Is it possible for Work,Personal contexts and for TextToSpeech,PredefinedMessages,Messages
7       features to be deactivated simultaneously? (yes/no):
8   -Is it possible for NightMode,LightMode features to be activated simultaneously? (yes/no):
9   -Is it possible for ContactHierarchy,Managing contexts to be activated simultaneously? (yes/no):
10  -Is it possible for ContactHierarchy,Managing contexts to be activated simultaneously? (yes/no):
11  -Is it possible for ContactHierarchy,Managing contexts to be deactivated simultaneously? (yes/no):
12  -Is it possible for ContactHierarchy,Managing contexts to be deactivated simultaneously? (yes/no):
13  -Is it possible for CSMode,Night,Offline contexts to be deactivated simultaneously? (yes/no):
14  -Is it possible for CSMode,Night,Offline contexts to be deactivated simultaneously? (yes/no):
15  -Is it possible for CSMode,Night,Offline contexts to be deactivated simultaneously? (yes/no):
16  -Is it possible for CSMode,Night,Offline contexts to be activated simultaneously? (yes/no):
17  -Is it possible for CSMode,Night,Offline contexts to be activated simultaneously? (yes/no):
18  -Is it possible for CSMode,Night,Offline contexts to be activated simultaneously? (yes/no):
19  -Is it possible for CSMode,Night,Offline contexts to be activated simultaneously? (yes/no):
```

Figure 4.3: Question asked by the tool for an example given by students.

This image shows that a total of 17 questions were asked for this example which already seems to be a lot of questions for a simple model. The second problem is that many questions are the same. In total we actually have 10 different questions with 4 of them asked 2 to 4 times. Another aspect that intrigues is the way those questions are asked. Lets compare the question on line 19 with the question on line 15 and the question on line 12 with the question on line 10. We notice that these question are comparing the same contexts and the only difference is that one question asks about the activation and the other one asks about the de-activation of those contexts.

This problem was due to the how mutations were created. For each mutation, a template of questions is used to generate questions to ask based on which contexts/features is used to create the question. The only thing that changes from one mutation to another is whether we ask those contexts/features can be activated at the same time or deactivated at the same time. This implies that some mutations have the exact same template of questions. If we add this to the fact that a connected pair can have multiple mutations that means that for a single connected pair there is always a chance that the same question is generated more than once. The new version that will be described later in this master thesis will implement a solution allowing use to avoid the question duplication problem.

4.3 Summary

In this chapter we presented the old version of the mutation testing tool. We saw what composed the program in terms of classes and how CFM models were put

into the program as input to be tested. We also dove more precisely into the code. As this old version was about to be used by a class of students, we thought it was useful in the second part to clarify some error messages and improve the capacity of the file parser to detect more errors in input files. The second part also identified different defaults in the program such as bad smells that must be avoided if we want a program to evolve or to fix coding errors. Finally, thanks to a model given by a group of students, we could identify more faults in the program by seeing that some questions were asked multiple times. In the next chapter we will analyze the survey given to the class and see what caused some issues found in the old version.

Chapter 5

Form and interpretation

As said previously, FBCOP was introduced in the course "Software maintenance and evolution" taught by Prof. Kim Mens. For this course, students have to gather in groups of two in order to write a context-aware program by first defining a CFM model. After creating their model during a laboratory class with the FeatureIDE [22] framework, they were asked to spend time during a second lab to encode their model on different *.txt* files and put them in the old version of the testing tool after being slightly modified as explained in chapter 4. From their experience with this version of the tool, we wanted to collect some feedback to know what actual users of the program think of this approach. To do so we created an online form you can find in Appendix A. This form was to be answered by each group of two students. Their answers were really helpful as they helped us to decide which direction this master thesis should take in order to improve the tool.

The first step was to design the feedback survey for students. This step was important as we had to define what are the objective of this form and select which questions are worth asking to the groups. The way each question is asked is also relevant because we cannot ask a lot of open questions to a user. This could lead to results that would be inconvenient to interpret later. Once the questionnaire was given to the class, the platform used to generate this doc allows us to access to all the students answer to the questionnaire as well as their model files

5.1 Preparing the survey

The form is composed of 3 distinct parts, each part of the form corresponds to a different step in the process of testing the CFM model. The first parts covers the model files, the second part is about the student's experience in using the testing tool and the last part is composed of questions regarding their results and the

output of the tool. At the end of each section, an optional open question allows each group to give some personal additional feedback.

The first part concerns the 3 .txt files that have been created by the group representing their CFM model, the context model and the mapping model. The goal was to have their global feeling about having to encode themselves their different parts of the model. We also tried to understand what are their needs regarding the current format. Maybe they want to use another encoding format to encode their model. We thought that maybe a better way would be to let them use a modeling language like UVL [23] That gives another syntax to write feature models. In the case that they would prefer UVL, we also needed to know if they where ready to spend more time of the lab installing a new program instead and spend less time encoding the model.

The second part was about their feeling and their experience about the tool itself. Data we wanted to collect concerned the need to use another format for the tool. Currently the tool works using a command line interface, the interrogation was then on the necessity to have a graphic interface and, if so, what kind of features would they want to see on this UI.

In the third part, we wanted to have an overview of the results given by the tool for each group. We wanted to know how many questions were asked to the group by the tool. Another interesting data was the number of recommendations that the tool gave corresponding to the answer to each question. We also wanted to know what proportions of recommendations were actually followed by the group and how useful those recommendations were. This section ends by an optional question allowing the group to report any bugs that could have been found during the use of the tool.

5.2 Interpreting results

The form was answered anonymously by a total of 8 groups which is not a lot but results are still useful as they can give us an idea of the general opinion. In this section, we will present some interesting results. Detailed results are available in appendix A.

5.2.1 Model files

A general tendency is that the current format was pretty easy to write. In fact it is also an aspect that was important for most of the students. The syntax is clearly

not adapted to have a good understanding of the model structure. Nevertheless the syntax is not the most clear, many students found that it was pretty easy to learn and that it allowed them to clearly show the hierarchy between features and context.

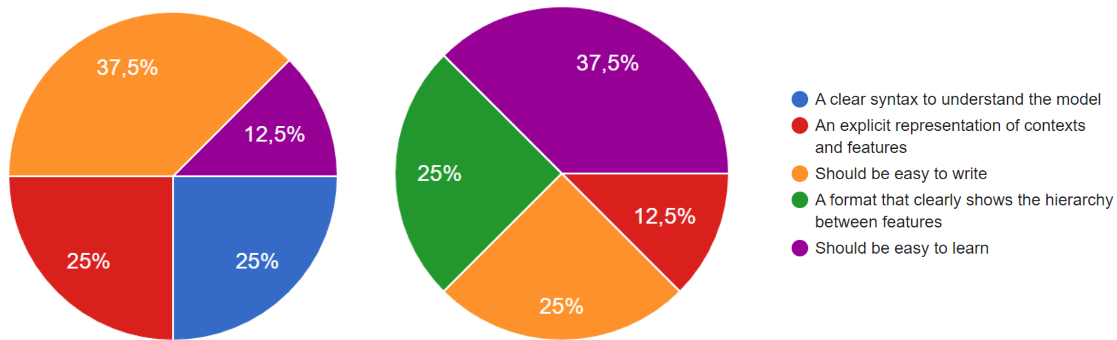


Figure 5.1: Question asked on the left: For you, what is/are the most important aspects that the input of the testing tool should have?

Figure 5.2: Question asked on the right: Which of these aspects were sufficiently supported by the current format?

5.2.2 Mutation testing tool

Generally users are more likely to prefer a GUI as they usually are more intuitive than a CLI. In this case, most of the students were satisfied of the command line interface and they were pretty clear that using a GUI for this kind of application was not really relevant. This could be due to the fact that possible interactions with the programs are not numerous as the main action was writing "yes" or "no" in the terminal.

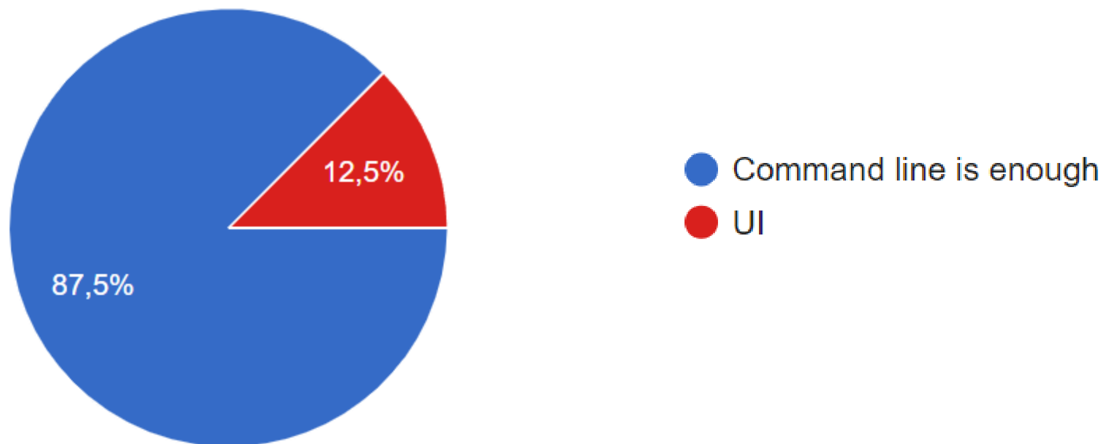
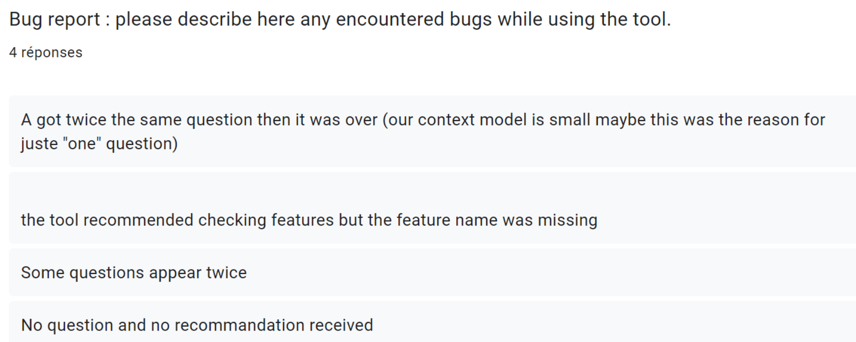


Figure 5.3: Question asked : Do you think the command line output format is enough or do you think the output could be improved with a UI?

5.2.3 Results

In the results section, we learned that the tool asks a maximum of 6 questions. 6 questions is still a good number, we don't want to have too many questions at the end of the testing process to keep the tool easy to use. But we noticed that some groups actually never received any questions. Actually, this part of the form helped us to understand something interesting, some groups actually were not able to use the tool correctly. For other groups, the same questions were asked multiple times. And sometimes, it was not even possible to reach the question generation step. For one of the groups, an error message appeared regarding a problem with a feature but without precising the name of feature that was causing this fault.


Bug report : please describe here any encountered bugs while using the tool.

4 réponses

A got twice the same question then it was over (our context model is small maybe this was the reason for juste "one" question)

the tool recommended checking features but the feature name was missing

Some questions appear twice

No question and no recommandation received

Figure 5.4: Results for the bug report question.

For the group receiving duplicate questions, this case was already mentioned in

4.2.2 and section 6.4 explains in detail what caused many other test cases to have the same question multiple times and provides a solution to solve this problem in the new tool. The second problem concerned a case where no questions were generated. This phenomenon is due to a context model containing only *Optional* constraints which are not currently supported. Chapter 8 will provide the implementation and the validation of a new mutation operator capable of solving this issue. The third case is a problem due to the parser of the tool which will be the first step in the re-implementation workflow

5.3 Summary

In the end, this form helped us to understand many interesting things. First, the current syntax is not perfectly usable. A future work could determine, what kind of syntax suits the best for a CFM model. As explained earlier UVL could be a solution but we need to extend the language to allow the model designer to specify a mapping model. The more urgent thing in the end is to solve all the problems that are present in the old testing tool. As explained in chapter 4, the old version of the program also contains different bad smells. The existence of these bad programming practice does not really allow us to easily solve those bugs. The solution resides in a complete re-implementation of the program but by still keeping the main ideas of the different steps in the workflow, and then improving the program further to overcome some of its current limitations.

Chapter 6

New mutation testing tool

Chapter5 gave us a general feedback for the first version of a mutation testing tool for CFM models. The end result of this feedback is that the syntax used to encode our models are considered good enough to keep it that way. The main problem remained the tool itself that didn't seem to work correctly. The goal then was first to fix the code before adding any new features to the program. Unfortunately, as illustrated in chapter 4, the code was not really clear and bad smells quickly deteriorated the habitability of the code. So instead of spending a numerous amount of time to fix a code that was not maintainable anyway, we decided to re-implement the entire program but still keeping the same workflow as the first version. This chapter will go through main parts of the new program by explaining them in detail and using the example defined earlier. This chapter will also be more oriented to the architecture point of view of the program. Appendix B shows the whole architecture of this program and each section will show which classes are involved in every step. Note that the workflow of the program stays the same as the one explained in the chapter treating about the older version.

6.1 File Parsing & Model creation

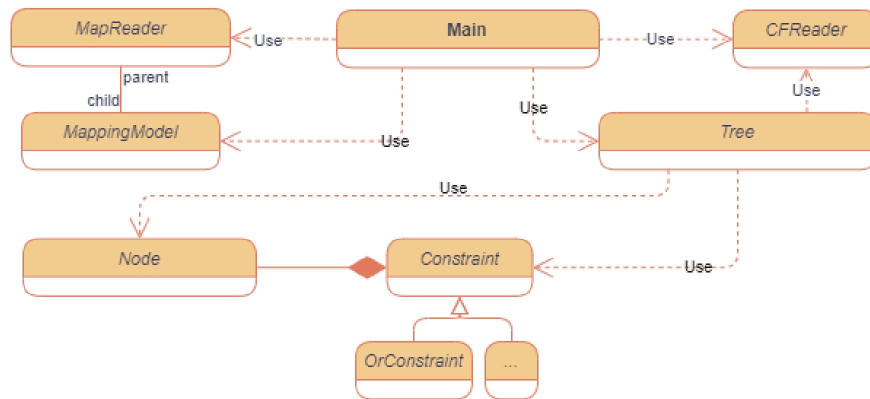


Figure 6.1: Class Diagram of the parsing module.

6.1.1 Context-feature model

The file Parsing module works pretty much the same way as the old version. The main difference is the use of different classes for each type of files. The *contexts.txt* file and the *features.txt* files are similar in their structure and the way to process them is the same so we will be using them with the **CFReader** class. In this case we differentiate the context file with the features by using two class attributes allowing us to know which kind of model we are currently generating. The purpose of the method is to simply go through each line of the file and at first look for any syntax errors or use of unknown node in the text file. If no errors are found then we instantiate a **Tree** object that will contain the context/feature model. With the model instantiated, all we have to do is to go through each line and create each node and constraint to add to the tree.

The **Tree** class is used to interact with a context/feature model. This tree is able to add or remove nodes from a given name, add a constraint group to the tree, find the parent of a node and is also able to be copied to create another tree. To search for a **Node** among all the nodes present in the tree, we defined a list containing every node present in the tree allowing us to search for a node in the tree with a linear complexity.

The **Node** class is basically here to carry data about a node in the tree, the node always has a parent unless it is a root node, a name and a list of children nodes. It is important to note that the **Node** class possess a *__hash__()* method

defined by python. This methods defines how an instance of a certain class can be hashed in order to use hashed structure. This method is crucial for the use of constraints as we will see in the next paragraph.

Constraint have a a hierarchical architecture, that means that every constraint are sub-classes of a super class calls **Constraint**. This class is basically an encapsulation of a set object. A constraint object is made to store in its set every node that it involves. There are two ways to instantiate a constraint, we can create it with an empty set or with a pre-defined set.

6.1.2 Mapping model

Now that we have completely defined how the context and the feature model are created from their input files, we can parse the third component which is the mapping model. To do so, we check the file line by line and we parse them like in the first two files. The difference is that this time we have another type of model to create. To build a mapping model, the first idea that comes into mind is to use a simple dictionary.

Unfortunately, this solutions does not really fit our expectations. The first problem is that each side of the mapping can be composed of multiple elements, that means that we do not have only one-to-one and one-to-many relationships but also many-to-many relationships. These kind of mapping is unfortunately not possible for a simple list because each key must be immutable to ensure that no collision occurs. A collision in key-value relation such as dictionaries is a repetition of the same key with a different value, this can lead to some problems as a query for a single value would lead to multiple answers.

The idea was then to use tuples as keys instead of list which are immutable data structure. This could work but the problem with tuples is that the order in which elements of the tuple are placed matters. Therefore, adding the tuple $("A", "B", "C")$ in the mapping would not be the same as adding the tuple $("B", "C", "A")$. This property causes a problem because when we have the same group of context in another line in the mapping file, but in a different order, that means that we will create a second entry in the mapping. On top of that, if I want to query a group of context knowing which contexts activates it, I cannot know in which order those contexts are put in the mapping model.

To cope with this problem, there is a new class called **MappingModel**. This class offers an encapsulation for a dictionary to which we can apply regular operations such as *get*, *set* and *add*. This encapsulations allows us to modify these

methods in order to use tuples with alphabetically sorted contexts. This is done by sorting the input list of contexts before querying the actual dictionary. Let's suppose the *mapping.txt* has the following mapping (the syntax here is simplified) $C1- > F1$, $C2- > F2$, $C1 - C2- > F3$, $C2 - C1- > F4$, then the resulting mapping model would look like $(C1) : [F1]$, $(C2) : [F2]$, $(C1, C2) : [F3, F4]$. If the caller wants to query $[C1, C2]$ or $[C2, C1]$ the output will then always be $[F3, F4]$

6.2 Sub-models

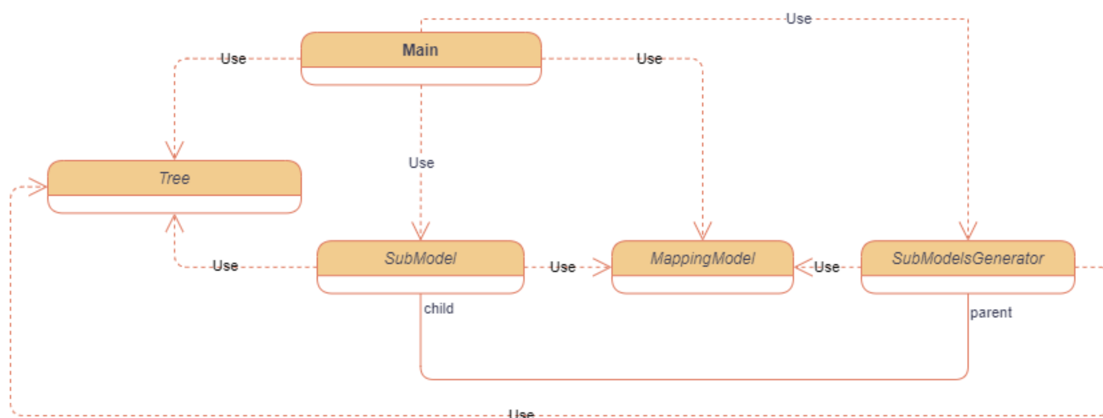


Figure 6.2: Class diagram of the sub-models generation module.

In chapter4 we introduced the concept of connected pairs, these pieces of a CFM model in which mutations will be applied. For simplicity reasons, we decided to change their name and simply call them sub-models. Sub-models have essentially the same role in the new program but , as we will see, we had to adapt their definition to be sure that the program would generate at least a few questions to the user.

To generate all our sub-models, we used 2 different classes, the first one is called **SubModelsGenerator**. This class uses the three component of the CFM model (context model, feature model and mapping model) to create a series of sub-models. Sub models are represented by the class **SubModel** which is implemented like a small CFM model with its context models, feature model and mapping model. Once these sub-models are created, they are stored a list and are ready to be used for the next step.

Before going through the entire mapping model we had to pre-process this model to separate each feature and make the model as atomic as possible. The

preprocessing of the mapping model uses the property defined in section 3.2 stating that each feature can be separated in a mapping model but we cannot separate contexts from each others. After that, we are ready to iterate through the whole mapping model and create sub-models.

Our first approach to create a sub-model was the one explained by A. Deckers, in this approach we considered that a sub-model was created whenever one child of a context activates at least one child of a feature and then the sub-model would be composed of both the child context and the child features as well as their parents and sister nodes. Once this sub-model was created, we would look at every sub-models that are already created. If we found a sub-model that had the same context model and the same feature model as the one that was just created, we simply merged the two mapping models without adding the new sub-model to the list.

The problem in this approach is that when a parent is taken in the sub-model, every children is also added in the sub-model regardless of the constraint. When children are in an *Or* or an *Alternative* constraint, it seems logical to put them together since they depend on each others. But if we have a *Mandatory* or *Optional* constraint, those children are completely independent from each other. If we take the example introduced in chapter 3, from the context *Localization*, we would also have *HighConnection*, *UserAvailability*, *Peripheral* and *ScreenOrientation* in the same sub-model even though they are not related to each other.

The solution used for to solve this issue consists in isolating *Mandatory* nodes and *Optional* nodes from each others. For these two constraints, we will only put the concerned child and its parent in the sub-model instead of putting it with other nodes with the same constraint. Thus giving a new interpretation of what a sub-model is : for each node involved in a mapping element, the sub models will contain the node with his parent and each other node(s) involved in the same constraint.

Now that the algorithm is completely defined, we have a list containing all our sub-models that are ready to be mutated. The example detailed in chapter 3 now contains 7 sub-models. You can find below a text representation of the the 7 sub-models created in our example. For each connected pair we have context models on first lines, then the mapping model associated and finally, the feature model.

Sub-models generated

```
UserAvailability -> Alternative(2){Available Busy }
{('Available',): {'FilterAvailable'}}
Feature -> Optional(1){FilterAvailable }

UserAvailability -> Alternative(2){Available Busy }
{('Available',): {'Alarm'}, ('Busy',): {'Mute'}}
Mode -> Alternative(2){Mute Alarm }

Peripheral -> Alternative(2){HasMicrophone HasCamera }
{('HasMicrophone',): {'Vocal'}}
MessageType -> Optional(1){Vocal }

Peripheral -> Alternative(2){HasMicrophone HasCamera }
{('HasCamera',): {'Picture'}}
MessageType -> Optional(1){Picture }

Context -> Optional(1){HighConnection }
Peripheral -> Alternative(2){HasMicrophone HasCamera }
{('HasCamera', 'HighConnection'): {'Video'}}
MessageType -> Optional(1){Video }

Context -> Mandatory(1){Localization }
{('Localization',): {'TranslateMessage'}}
Feature -> Optional(1){TranslateMessage }

ScreenOrientation -> Alternative(2){Portrait Landscape }
{('Portrait',): {'Vertical'}, ('Landscape',): {'Horizontal'}}
View -> Or(2){Vertical Horizontal }
```

6.3 Mutations

For the testing of a CFM model, a mutation on a sub-model would mean a simple modification of a constraint inside this feature model. Since the algorithm previously defined for the sub-model generation allows us to have a single constraint for each node, what we have to know for each sub-model is on which constraint we will apply a mutation and which mutation we are going to apply on this constraint.

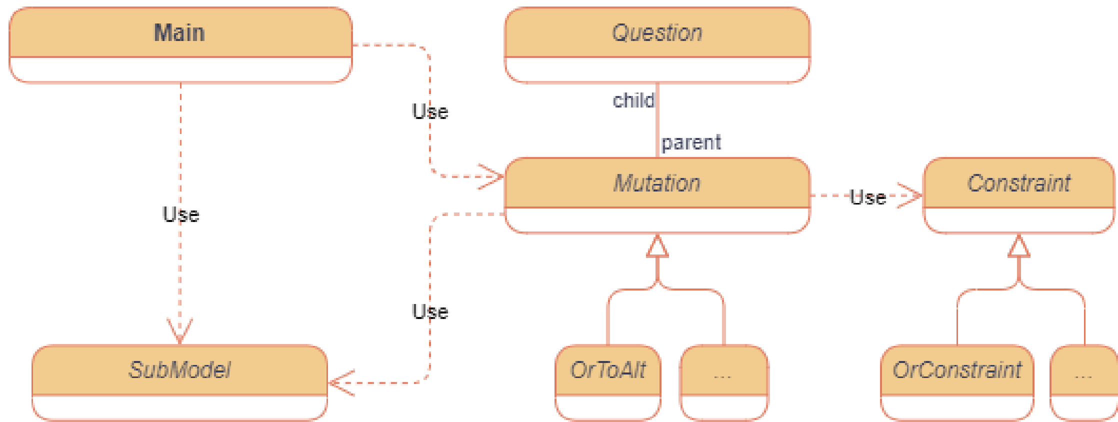


Figure 6.3: Class diagram for the mutation module.

The mutation modules is inspired by the visitor design pattern [24]. This pattern is adapted to our situation because we have different modifications (mutations) that can be applied a series of object (sub-models) and we have to visit each objects to know what modification can be applied. Our implementation of a design pattern is a little bit different than the original one. Indeed, each visitor corresponds to a mutation and we wanted to avoid creating a mutation object for every visited sub-model. To do so, we used static methods in each mutation classes. A static method is a method that will be associated to a class rather that to an object. Luckily, python helped us to achieve this by providing us the method *__subclasses()__* which returns a list of all classes that are inheriting from a given class. In our case, each mutation is inheriting of the **Mutation** class. Therefore all we have to do generate our mutants, is to iterate through each sub-model and for each one of them, use the *is_applicable()* method and start the mutation if the later returns `True`.

To see if a mutation is applicable or not, we have to respect two different rules. First, to stay in the logic of the program defined by A. Deckers, we have to respect the hierarchy defined in figure 4.2. The constraint at the top of the hierarchy is the *Alternative* constraint as this constraint is the most restrictive in terms of the number of activated nodes it can accept at the same time.

The second reason is that some mutations do not have any effect on the set of valid configuration of a sub-model if they are not applied correctly.That means that you cannot always apply a mutation on a constraint even if it respects the constraint hierarchy. As an example let's take the following mutant, here we applied an **AltToOr** mutation on the left side. Since the right size is an *Alternative* constraint, the model can only activate *Alarm* or *Mute* but not both at the same

time forcing the new *Or* constraint to behave like an *Alternative*.

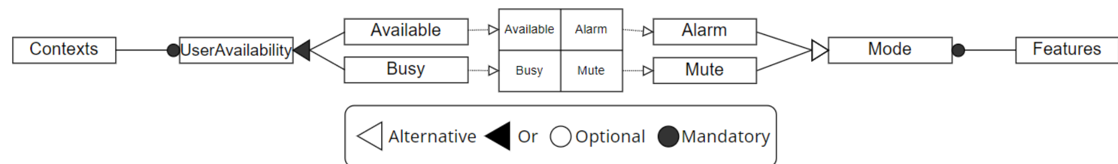


Figure 6.4: In this example, an **AltToOr** mutation on one side is not relevant.

This is a good example of why mutations are not possible everywhere in the model, since the new constraint behave like the original one, the mutation becomes useless. In that case, the solution is to apply the mutation on both side of the sub-model. Unfortunately there is no general case that helps us to quickly define how and where a mutation can be applied so we have to handle this kind of rare cases one by one.

Finally, if the mutation turns out to be applicable, the **Mutation** class contains a function that applies the mutation on a given model. The classical way is to take a copy of the sub-model and then change the type of constraint. This is done by simply creating a new instance of a constraint and fill the encapsulated set of the new constraint with all the nodes that are the old one.

6.4 Question generation

In the new version, questions are generated together with each mutation since a mutation is always linked to the questions. In the new version, a question is a object of the class **Question**. For each questions we keep track of the mutant used to create it and the expected answer which is the answer for which recommendations will be given. We also keep the name of each contexts and features contained in the question and of course, a string containing the phrase that will appear to the user.

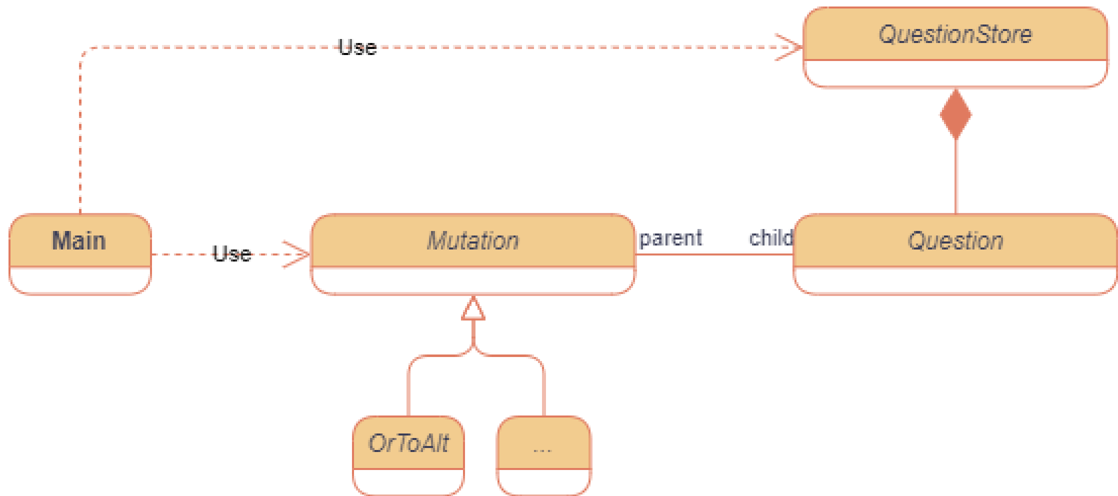


Figure 6.5: Class diagram of the question generation module.

In the general case, a questions will only present to the user all contexts and features implied and ask him if they can be activated together or not. But here again, there are some exceptions. The first exception occurs when there is a mutation that aims into transforming any constraint to an *Mandatory* or an *Optional* constraint, in this case we only ask if the node involved can be activated or not. The second case is for an **OrToOpt** mutation. For this particular mutation, what we want to know is not if a set of contexts/features can be activated but rather if they can be de-activated [6]. In the first case, the question is directly generated in the subclass but in the second case, we simply added a condition that recognize the mutation applied in a question generation method defined in **Mutation**.

Let's go back to our messaging application example (see 3). After creating our sub-models, we passed through the visitor pattern to generated mutations and from mutations we generate the following questions:

```
Generated questions
1 Is it possible for Available, Busy contexts and for Alarm,
  ↪  Mute features to be activated simultaneously?
2 Is it possible for Available and Busy contexts to be activated
  ↪  simultaneously?
3 Is it possible for HasMicrophone and HasCamera contexts to be
  ↪  activated simultaneously?
4 Is it possible for HasMicrophone and HasCamera contexts to be
  ↪  activated simultaneously?
5 Is it possible for HasMicrophone and HasCamera contexts to be
  ↪  activated simultaneously?
6 Does Localization have to be activated in any configuration?
7 Is it possible for Simple and Complex features to be activated
  ↪  simultaneously?
8 Is it possible for Mobile and Computer contexts to be
  ↪  activated simultaneously?
```

The only problem here is that question 3, 4 and 5 are exactly the same. This phenomenon can appear for different reasons, a same sub-model can pass through multiple mutations that generate the same question multiple time or two different sub-models that have a part of the left side (contexts) or the right side (features) in common passed through a mutation that used this common part. In our example, questions 3, 4 and 5 have a contexts model in common and were applied an **AltToOr** mutation that uses the left side of the sub-model.

To avoid the user to receive the same questions repeatedly, we defined a new structure adapted to contain each question, the **QuestionStore** class. This structure encapsulates a list of questions, if a question with the same string is already present in the list, it is not added. But since some questions coming from different sub-models or mutations can have the same string, they can have different recommendations and the store makes sure that we have all recommendations stored. The process to generate recommendations is explained in the next section. We now removed two questions from the list above and avoided asking repeated questions to the user. Chapter 7 will explain how the questioning step can be improved using the right starting question with a decision tree.

6.5 Recommendations

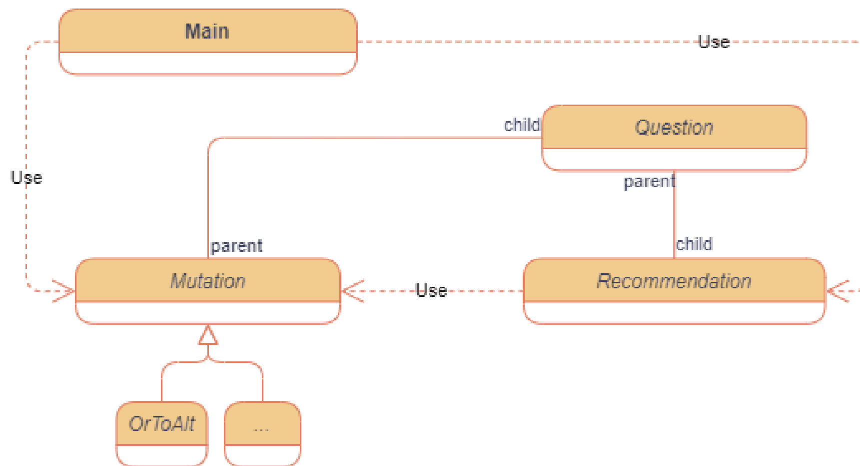


Figure 6.6: Class diagram of the recommendation module.

For each generated questions we keep track of an answer field. If the user answers the question with the value of the field, that means that a potential flaw is detected in the CFM model and a recommendation is generated explaining how the user should modify the model. This recommendation is defined by the **Recommendation** class. This class uses the contexts, the features and the mutant implied in the question to create a string that will be displayed to the user. As an example here is what an interaction with the tool could look like for a user who receives recommendations in the example model.

```
Is it possible for Busy, Available contexts and for Alarm, Mute features to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for Busy and Available contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> yes
Is it possible for HasMicrophone and HasCamera contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Does Localization have to be activated in any configuration?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for Vertical and Horizontal features to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> yes
Is it possible for Landscape and Portrait contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> yes

There might be some errors in the model. Here is a list of recommendations that could fix them.
        Try to modify the constraint from the following context(s) : Busy Available with the following mutation : Alt-To-Or
        Try to modify the constraint from the following context(s) : Localization with the following mutation : Man-To-Opt
        Try to modify the constraint from the following context(s) : Landscape Portrait with the following mutation : Alt-To-Or
```

Figure 6.7: Example of how the recommendations are shown to the user after (randomly) answering to every questions.

This example shows what an interaction with the tool looks like from the user point of view. After generating every questions, the tool asks the user to provide an answer to each one of them. Every time a question receives an answer from the user, the tool process this answer and decides whether it has to generate a recommendation or not. Once all questions are answered, the user receives a summary of all recommendations that have been generated.

6.6 Summary

In this chapter, we introduced the new version of the FBCOP mutation testing tool. This is a re-implementation of the old version introduced in chapter 4. The purpose of this re-implementation was first to solve bugs that were found in the old version at first. But since the old version contained bad smells, we finally decided to restart the development from scratch but keeping the same workflow as the first version. This re-implementation of the tool was also an opportunity to make the program more habitable and to facilitate the addition of new constraints and mutations.

During the development we also had to redefine the concept of connected pairs that we now call sub-models. The mutation module keeps the same logic but can now evolve more easily thanks the visitor design pattern. Finally, we also went through the step of generating questions from those mutations and we explained how recommendations were displayed according to the answers of the user.

Even if the program seems to work correctly, we would like to explore the possibility to improve the way questions are asked. To improve this step we will now consider a solution based on a decision tree.

Chapter 7

Improving questions generation

In the previous chapter, we presented the newest version of our mutation testing tool. Our tool works by mutating sub-models and from those mutations it generates different questions whose answers will tell us whether or not a design error is present in the CFM model. In section 6.4, we showed that when questions were generated it was possible to have the same question appearing multiple times to the user. The solution used for this issue was to store those questions in a set to avoid repetitions. At this stage, the messaging applications example gives us the following questions :

Generated questions after repetitions filtering

```
1 Is it possible for Available, Busy contexts and for Alarm,
  ↪  Mute features to be activated simultaneously?
2 Is it possible for Available and Busy contexts to be activated
  ↪  simultaneously?
3 Is it possible for HasMicrophone and HasCamera contexts to be
  ↪  activated simultaneously?
4 Does Localization have to be activated in any configuration?
5 Is it possible for Simple and Complex features to be activated
  ↪  simultaneously?
6 Is it possible for Mobile and Computer contexts to be
  ↪  activated simultaneously?
```

After removing repeating questions, we asked ourselves if it was possible to create an algorithm that uses previous answers to anticipate the answer to other questions. In this chapter will present 2 different algorithms that use a decision tree in order to decide which question should be asked next according to the previous

answer. We will also use a propagation algorithm that will guess the answer to other questions from the answer given to the last question.

7.1 Theoretical foundations

To know for which question we can anticipate the answer we have to look at a question from a different angle. In section 2.1.2 we talked about transforming a CFM model into a boolean proposition to check for distinguishing configurations. It is actually possible to do the same things with questions coming out of the tool. A question is always composed of one or more nodes that can be activated or deactivated. Each node can be seen as a boolean variable, when the node is activated, then the variable is `True` and when the node is deactivate its value is `False`. By making this comparison, we can consider that most questions are then asking if it is possible to have all those variables set to `True` simultaneously and therefore asking if the boolean conjunction of those variable is satisfiable.

7.1.1 Using subsets for anticipations

We can now derive the following property for boolean conjunctions. Let's consider a question $\mathcal{Q}$ asking if each node of the set of nodes $\mathcal{N}_{\mathcal{Q}}$ can be activated simultaneously. If the user's answer to $\mathcal{Q}$ is "yes" then there is a valid configuration in which:

$$\bigwedge_{n \in \mathcal{N}_{\mathcal{Q}}} n = \texttt{True}$$

Since all nodes in $\mathcal{N}_{\mathcal{Q}}$ can be set to `True`, then any subset of $\mathcal{N}_{\mathcal{Q}}$ can have all nodes set to `True`. As an example we can take the first questions on the list above, we have $\mathcal{Q}_1$ = "Is it possible for Available, Busy contexts and for Alarm, Mute features to be activated simultaneously?" and $\mathcal{N}_{\mathcal{Q}_1} = \{Available, Busy, Alarm, Mute\}$. If the users answers "yes" to this questions that would mean that there is a valid configuration where all variables in $\mathcal{N}_{\mathcal{Q}_1}$ are activated. If we take the second question, we have $\mathcal{Q}_2$ = "Is it possible for Available and Busy contexts to be activated simultaneously?" and $\mathcal{N}_{\mathcal{Q}_2} = \{Available, Busy\}$. We observe that $\mathcal{N}_{\mathcal{Q}_2} \subseteq \mathcal{N}_{\mathcal{Q}_1}$. We can then deduce that for any configuration where

$$\bigwedge_{n \in \mathcal{N}_{\mathcal{Q}_1}} n = \texttt{True},$$

we also have

$$\bigwedge_{n \in \mathcal{N}_{\mathcal{Q}_1}} n = \texttt{True}$$

That means that the answer to $\mathcal{Q}_{\in}$ is also "yes". We will call this property the Anticipated Subset Evaluation Property (ASEP).

The ASEP can be applied to improve how our questions can be asked. If the answer given by the user to a question using certain set of node is "yes", then we just have to check for other questions that use a subset of those nodes. We will then be certain that the answer to these question will also be "yes" and we won't need to ask this question. And if one of those questions implies the existence of an error in the model, then corresponding recommendations will be generated.

7.1.2 Using super sets for anticipations

But what if the user's answer to a question is "no"? The ASEP would certainly not be applicable. If we use the boolean analogy, it would mean that there will always be at least one deactivated node in the set used for the question. It is then impossible to make any assumptions about subsets of the nodes set. Luckily, we are able to change the ASE property to build its reciprocal. Instead of looking for questions that use a subset of nodes used in the question, we will be looking for questions that use a super set of these nodes. In other words, if the user said "no" to a question $\mathcal{Q}$, that would imply that

$$\bigwedge_{n \in \mathcal{N}_{\mathcal{Q}}} n = \texttt{False}$$

Since a conjunction that returns `False` has at least one of its variable set to `False`, using the same conjunction with additional variables will always be evaluated to `False`. This is why looking for a super set in case the answer is "no" could also help to anticipate the answer to some questions. If we retake the example above, we have that $\mathcal{N}_{\mathcal{Q}_1}$ is a super set of $\mathcal{N}_{\mathcal{Q}_2}$. Therefore, if we ask $\mathcal{Q}_2$ first and the user answers "no", we are able to infer the answer "no" to $\mathcal{Q}_1$.

7.2 Questioning strategies

The next section will present two new methods used to ask questions to the user. In total, the user is able to chose between 3 different algorithms that will define the way questions are asked. For that, we would need a design pattern that decides which algorithm to use at run-time in order to give the choice to the user. The most adapted design pattern for this is the strategy design pattern [10]. Figure 7.1 shows how the strategy design pattern was implemented in the tool to allow the

user to chose which algorithm he wants to use to receive questions.

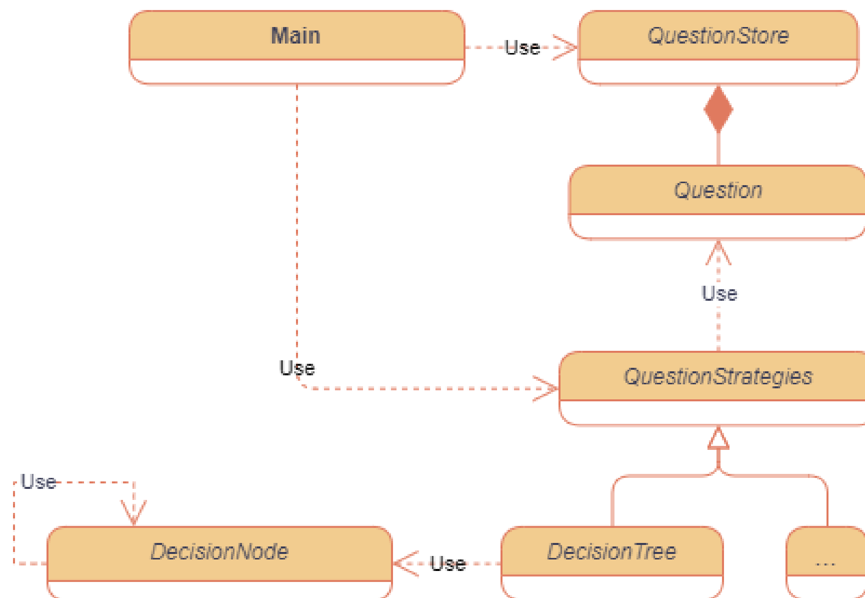


Figure 7.1: Class diagram of the question strategy module.

The chosen design pattern allows us to now have 3 different strategies. The first strategy will not really be discussed as it is the most basic one: we ask every question as they are after removing repeating questions and use the answer provided by the user to simply generate recommendations if needed. The second algorithm uses previously answered questions and tries to anticipate the answer to other questions based on what nodes each question is using. Finally the third strategy works the same way as the second one but makes a summary of anticipated questions to the user in order to let him confirm that predictions are correct.

7.2.1 Decision tree

To apply the ASEP, we decided to use a decision tree. A decision tree is a data structure allowing us to use previous answers given by the user in order to determine which question will be asked next. If the user decides to use this strategy, once all the questions are stored in the question store, a decision tree will simulate every possible series of answers coming from the user.

Before creating the tree, we have to define a default order in which questions will be asked. To decide that, we use the competent programmer hypothesis [13]. Thanks to this hypothesis we can imagine that the majority of the constraints are

correct in the model and a minority of them are wrong. Since most of the questions detect an error if the answer is "yes", we can state that the user is more likely to answer "no" to a question. We now have to think how we can use the answer "no" to anticipate a question (and therefore prune it from the decision tree). As explained earlier, if the answer is "no", we will look for a super set to prune a question and super sets can only be found if we have a question using a greater number of nodes. Our implementation will then use questions that use either context nodes or feature nodes at first and then questions that use both contexts and features.

Once we have a predefined order to ask questions, we can build the decision tree. The decision tree is built recursively in the **DecisionNode** class. Each node contains a question to ask to the user but also keeps track of which questions are already asked as well as their answer and which questions still have to be asked. Each node has two children node, if the answer given to the user is "yes" then the next question to ask is the question on the left child, if the answer is "no" then the next question is in the right child. Since each child node simulates a different answer for the parent's question, we have to propagate this answer to the questions that are already asked. The **DecisionNode** class contains two methods for that, one to propagate "yes" and the other one to propagate "no". To chose which question is asked in a given node, we always take the first question in the list of unasked questions.

With the decision tree built, the first question can now be asked to the user and according to his answer we will then go to the left node or the right node until we reach a node where every question received an answer.

7.2.2 Decision tree with feedback

In this strategy, the user has the possibility to change some questions predicted by the decision tree explained in section 7.2.1. Once the user answered to all the questions that were asked, the tool shows him all the questions that were predicted from his answers. Then with those questions, he has the choice to change the answer from some of those questions by specifying which one are wrongly predicted.

To allow this strategy to work, the **DecisionNode** class need to take track of which question has a predicted answer and for each one of them, what is the predicted answer. During the propagation of each possible answer, the dictionary keeping track of those questions is updated by adding the predicted question.

7.3 Summary

This chapter concluded the presentation of the new version of the mutation testing tool. We first explained a property of boolean conjunctions that we can apply on the group of nodes used for a generated question. Then we used this property to improve the way the program asks questions to the user. This improvement is made thanks to a decision tree that will simulate how the user can successively answer to all questions. Depending on which answer the user gave, we can use the property defined at the beginning of the chapter to prune some questions from the tree because the answer given by the user allows us to anticipate the answer to other questions.

In the final chapter of this master thesis, we will discuss different test cases and see how the program worked with them. We will finally present some improvements that can be made to the actual program and try to propose potential solutions that can be implemented to develop the subject.

Chapter 8

Validation

Now that all changes made for this master thesis have been presented, this chapter will analyze different results of the new mutation testing tool. We will first start with a validation of the new version by using different test cases that were available for us by exploring different aspects that differ from the old version. We will also show that the new version has a good scalability by enriching the set of mutations. The second part will present different threats to the validity of this master thesis by exploring different weaknesses. Finally, we will end this chapter by presenting different future works that can be done to improve the testing tool that we just re-created.

8.1 Validating the re-implementation

8.1.1 Methodology

In order to verify that the model worked correctly, we needed to have access to different models that could be tested. Luckily, as explained in chapter 5 when students filled the form during the lab they were asked to provide their*.txt* files. This ensured that if they encountered a problem while using the tool, we could see by ourselves what was the problem in order to fix it.

8 students answered the form so we could have 8 different models available. But we noticed that 2 groups submitted twice their model files? In the end we actually have 6 test cases + the test case defined in chapter 3. In order to make each group anonymous and for simplicity reasons, we decided to rename our test cases *testX* where "X" is the number of the model going from 1 to 6 and the messaging application will be called *basic_example*.

8.1.2 Analysis

Comparing the new version with the old version

To compare the new version with the old one, we decided to use 3 different metrics. The number of questions asked to the user, the number of sub-models generated and the number of mutations.

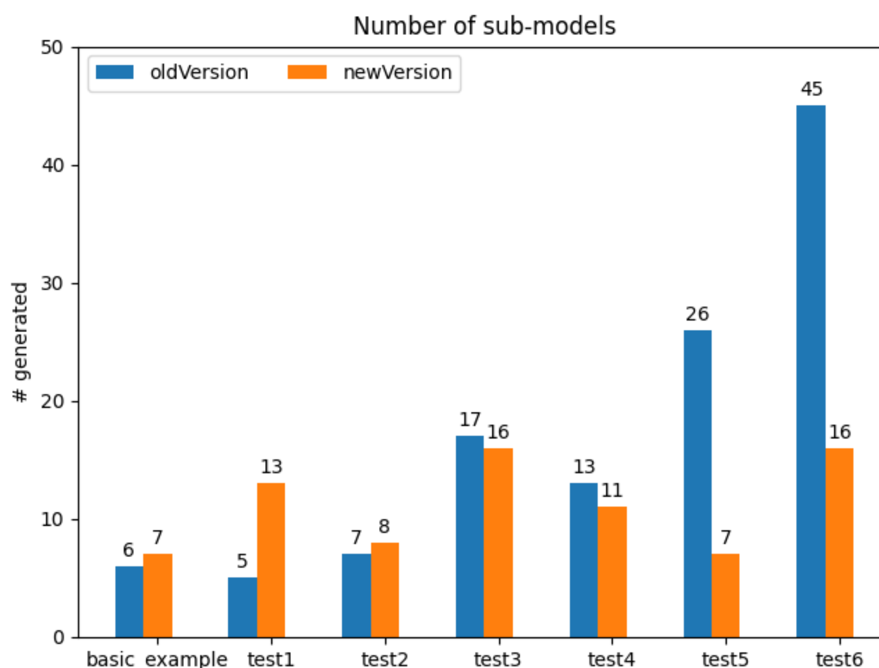


Figure 8.1: Comparison of the number of sub-models created in both versions of the testing tool.

In the old version, *test5* and *test6* had a really important number of sub-models with respect to the number of mapping element they had (respectively 26 and 45). Now in the new version we reduced the number of sub-models. *test5* has now 7 sub-models which correspond to the number of mapping element present in the model. In *test1* the number of sub-models almost tripled, this is due to the new method used to create them. The old version of the algorithm allowed to absorb more mapping to the element because when a node was taken into a sub-model, every sister nodes were taken with him. This could cause a lot of nodes to be part of the same sub-model decreasing their number.

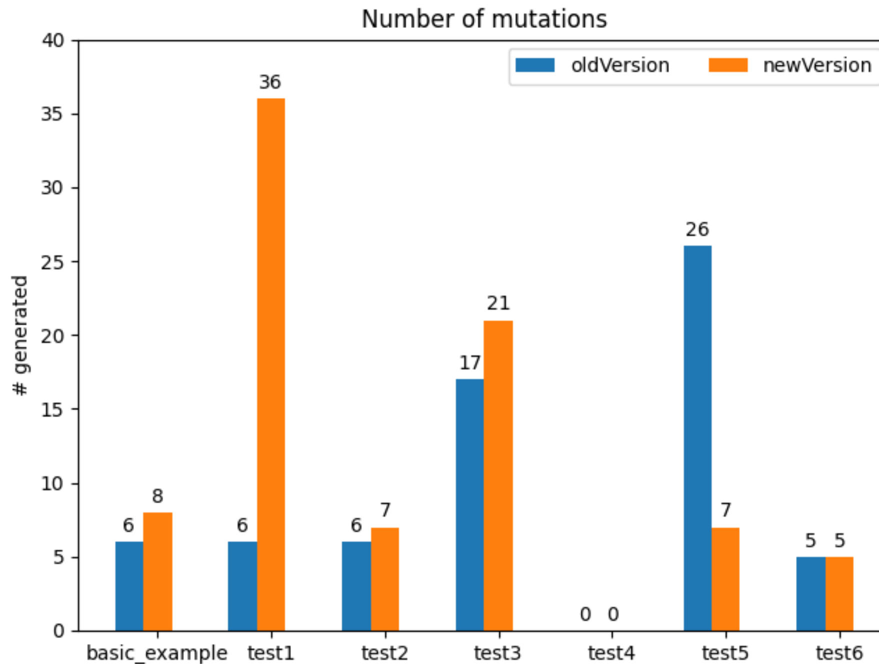


Figure 8.2: Comparison of the number of mutations in both version of the testing tool.

In the figure above, *test1* has a number of generated mutants almost 3 times higher than the number of generated sub-models in the new version. In the mutation module, each sub-models tries to make every mutations. That means that for a model with a number k of sub-models knowing that we implemented 4 mutations we can have up to $4 * k$ mutants. That means that *test1* can have a maximum of 52 mutants. The reason why this test has more mutants than any other test is probably because its sub-models respects more some mutations conditions than other models. *test4* does not have any mutations. As explained in section 5.2.3 a group did not receive any questions when they started the program. By inspecting their context model, we noticed that every context is in an *Optional* constraint and no mutation is currently designed to mutate this constraint. In the new tool, it was not implemented at first but we will use this case later to demonstrate the scalability of the new program.

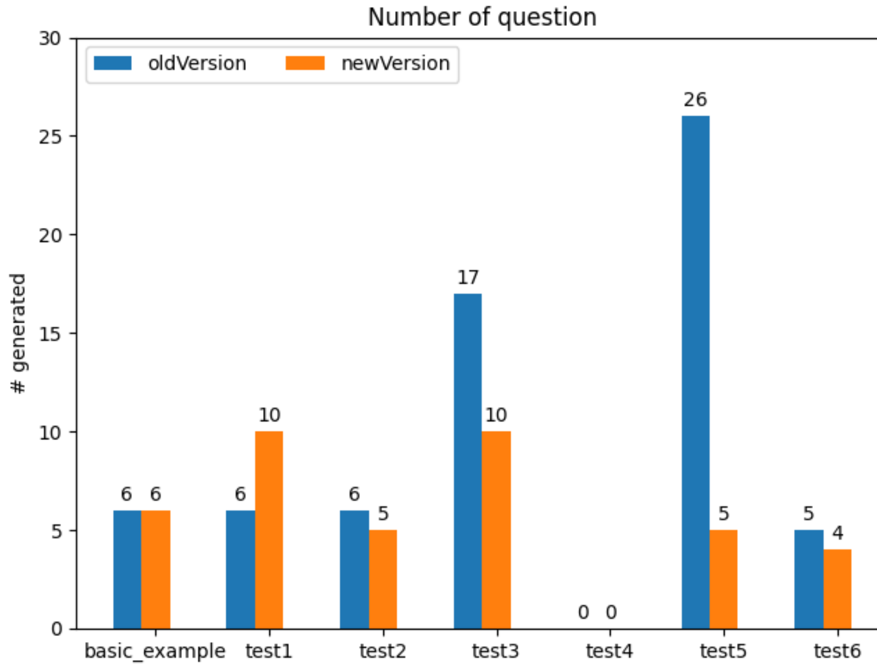


Figure 8.3: Number of questions asked to the user foe each test.

Since there is a 1 to 1 relation between the number of mutants and a question object, It is normal to see that for each test, the number of questions is never higher than the number of mutation. The difference between the number of mutations and the number of questions can actually measure the effect of the question filter explained in section 6.4. The number of questions in *test1* is now 10 so it decreased to a third of the number of mutations, that means that a lot of generated questions are actually the same, which is more likely to happen when we have a high number of mutants. In *basic_example* we can see that the number of questions is 6 and we had 8 mutations as explained in section 6.4.

Testing the message application model

Since chapter 3, we followed the journey of a messaging application model that was testing with the new tool created for this master thesis. In this example we purposely inserted three design errors and we want to see if the new tool is able to find them. The three errors are the *Localization* context that should be optional,*HasMicrophone* and *HasCamera* should be in an *Or* constraint and finally, *Vertical* and *Horizontal* features should be in an *Alternative* constraint. To be able to find those errors we put ourselves in the shoes of a model designer and we answer each question in the most realistic way possible.

```
Is it possible for Available and Busy contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for Available, Busy contexts and for Mute, Alarm features to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for HasMicrophone and HasCamera contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> yes
Does Localization have to be activated in any configuration?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for Vertical and Horizontal features to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no
Is it possible for Portrait and Landscape contexts to be activated simultaneously?
Please Answer with one of the following : "yes", "y", "no", "n"
>>> no

There might be some errors in the model. Here is a list of recommendations that could fix them.
        Try to modify the constraint from the following context(s) : HasMicrophone HasCamera with the following mutation : Alt-To-Or
        Try to modify the constraint from the following context(s) : Localization with the following mutation : Man-To-Opt
        Try to modify the constraint from the following feature(s) : Vertical Horizontal with the following mutation : Or-To-Alt
```

Figure 8.4: Interaction with the new tool with the use case defined in chapter 3.

As we can see, our answer receive 3 different recommendations that correspond exactly to the 3 errors that were put in the model. To remove them we only have to apply these recommendations in order to correct the model.

New version's scalability

One of the main goal of the re-implementation of this new version was to improve the scalability in order to easily add mutations and constraints. As previously explained, a group of students did not received any questions because the program does not support mutations on *Optional* constraint. To fix this problem we will demonstrate how to add a new mutation in the program by adding a simple **OptToMan** mutation. To add a mutation we must first create a class that inherit of the **Mutation** class. Then only two methods are needed to be implemented : *is_applicable()* and *visitSubModel()*.

For the first method we will keep things simple, we will just check if the sub-model that we are visiting contains an *Optional* context. To apply the mutation we must now implement the *visitSubModel()* method. The first step to apply a mutation is to copy a model in order to have a new one that we can modify without interfering with already existing sub-models. In this new sub-model, we now have to find which context sub-model contains the constraint that has to mutate. The mutation is then applied thanks to the *applyRegularMutation()* method in the mother class. Finally, from the mutant, we can generate the question to ask. The questions has the same pattern as the **ManToOpt** mutation but the expected answer changes because now the context must be permanently activated. The implementation of the **OptToMan** is detailed in Appendix C.

Improving the questioning algorithm

To validate the new questioning algorithm, we compared how many questions the user was expected to receive when choosing algorithm or the other. When we do not use any optimization the user will always be asked every question in the question store. But if the user chose to optimize the questioning, we might expect to have some questions pruned according to the answer that we provided. For each test case, we built the entire decision tree to check every possible path and calculate the average depth of the tree.

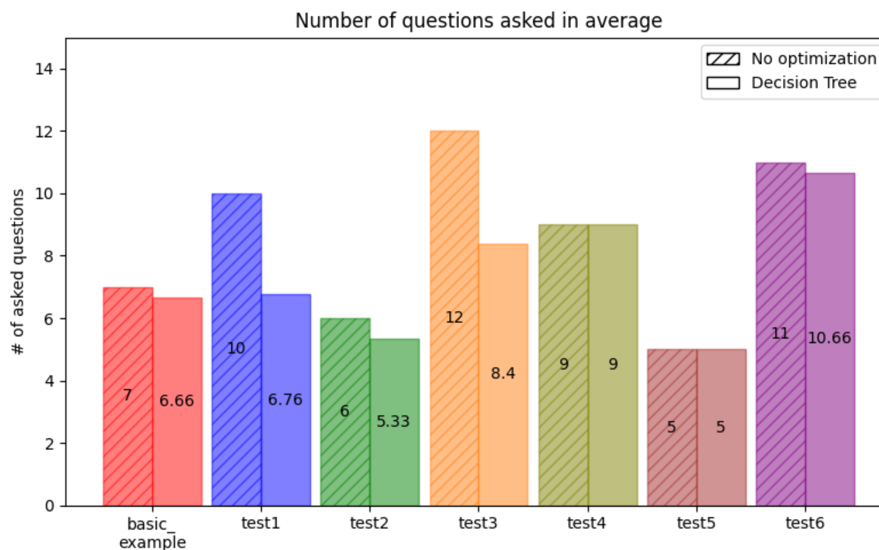


Figure 8.5: Number of expecting questions for according to the chose algorithm for every test case.

In this graph we can also see the effect of adding the **OptToMand** mutation in the system. We observe that *test4* has now 9 questions but no pruning. In this case this is because each context is *Optional*. Since the *Optional* constraint can only contain one node, the property explained in 7.1.1 will never be used. For the other subsets we notice that for many examples the pruning does not allow to remove a great number of questions in average.

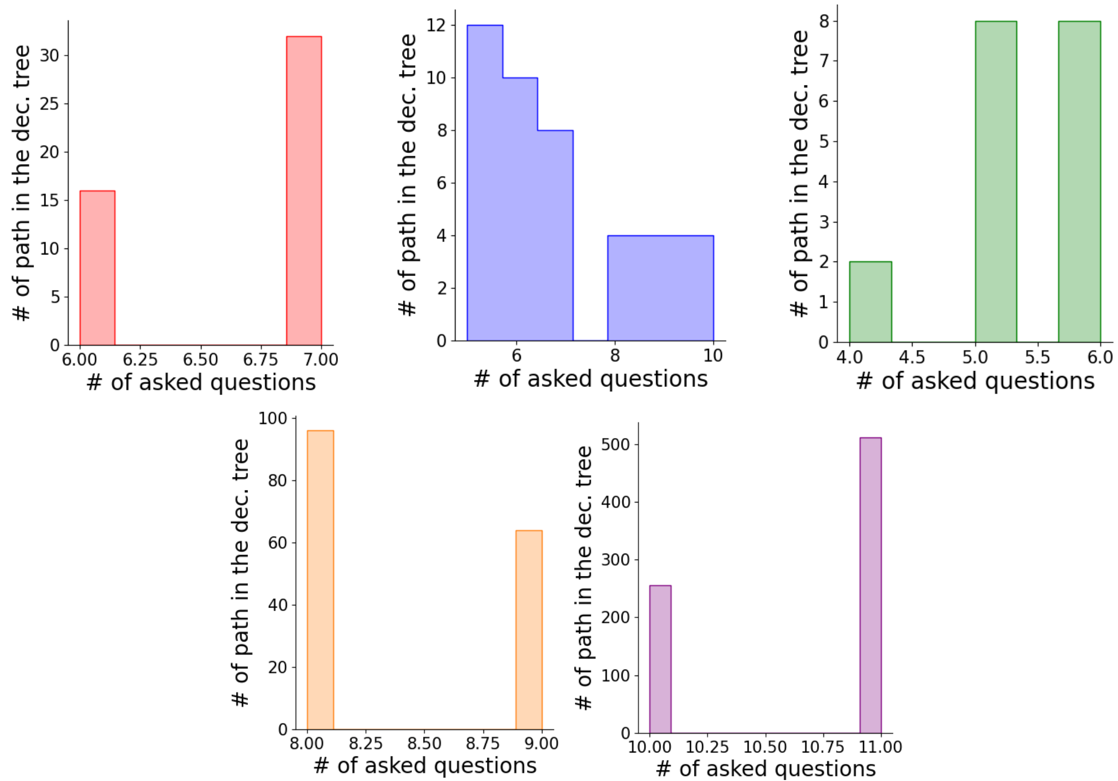


Figure 8.6: Distribution of the number of questions asked to the user for each test case.

By looking at the distribution of those paths, we notice that for *test1* that the majority of path length is around 5. That means that the user is the most likely to answer to only half of the questions and the algorithm can deduce the answer to the other half which is a great improvement . We can also notice in *test3* that every path is of length 8 or 9 so the number of questions asked always decreases by at least 25% when using the decision tree showing how efficient the new algorithm can be. For the *basic_example* there is only one question that can be prune. The pruning happens if we answer "no" to question 2 of the list (see 7). On average we were able to prune up to 19% of the questions thanks to the decision tree.

In the end, for many models, the number of questions that we are able to prune according to answers to previous questions is relatively low but there are still some models that can always prune questions regardless of the answer. The fact that almost 20% of the questions can be saved makes this algorithm still interesting to use.

8.2 Threats to validity

Test cases that are used during this validation process are defined for the same exercises as well as our use case, the creation of a smart messaging application. The fact that these models have the same purpose does not necessarily means that they have to be the same, they can have some similarity in their conception. These similarities can then lead to poor coverage of constraint combination in the test suite.

8.3 Future work

8.3.1 Adding new mutants operators

The goal of re-implementing the testing tool, was first to fix issues found in the old version but was also to allow someone to introduce new mutants operators. This was done thanks to the visitor design pattern [24]. For now, only a limited number of mutants are available but thanks to the design pattern, nothing stands in the way of adding new mutants to the system. We could also imagine adding a new kind of operator. A. Deckers. proposed to add mutants based on the mapping model. One possible operator could be the **Swap** operator. This operator would swap two features or two contexts in the mapping component in a sub-model [6].

To add a new type of constraint in the tool, we just have to create a class for this constraint that inherit of the **Mutation** super class. This super class would then let us implement two static methods. The first one is called *is_applicable()* and takes three arguments : a sub-model, a context constraint and a feature constraint. This method is here to check if the sub-model that is being visited respects all the requirement to apply the mutation that we are defining. The second method is the *visitSubModel()*. It takes the same arguments as *is_applicable()* and is here to apply the desired mutation to the sub-model in order to generate the question.

8.3.2 Cross-tree constraints

During this master thesis we only talked about constraint that contained a parents node and one or more children nodes. In reality, feature models can use constraints that concern nodes that are not directly related in the model. So we can define constraint between nodes everywhere in the tree. We could also imagine a constraint that connects nodes from the constraint model and the feature model. Since this would introduce new constraints in the system, we could think of adding new mutations applied to this kind of constraints.

One way to implement these constraints would be to create a new class that inherits from the **Constraint** class. This class would then contain every node of this constraint. Then since the new constraint is not part of any component of the CFM model architecture, we could create a 4th component that works independently from the 3 others. If we want to mutate these constraint, we would probably have to have a visitor design pattern adapted to them as the current one is made for sub-models.

8.3.3 Xml files parser

In the survey given to students, some student pointed the fact that the syntax was not easily readable at that it did not allow to understand the model correctly. Another problem is also that for a bigger model, it can be pretty hard to encode it without making any mistakes, especially if a node is contained in different constraints.

In the course of Software Maintenance & Evolution, students are usually asked to first visually create their feature and context models with a tool called FeatureIDE [22]. This tool provides a way to just create feature models for feature-oriented domain analysis. The advantage that this tool gives us is that each model is saved in an Xml file that contains each node but also all constraints used in the model. The solution would then be to implement a script able to parse these Xml files to create both models inside the tool's code base. The only problem remains the mapping model, FeatureIDE does not provide any tool to design a mapping model from two tree models. A possible solution to cope with this problem would be to keep the *.txt* file for the mapping but it would need to have a more understandable syntax. Another solution would be to use appropriate UI libraries to let the used define his own mapping model in a more visual way.

8.3.4 OrToOpt pruning

One case that was not taken into account in our questioning algorithm, is the case where the mutation associated to the question is an **OrToOpt**. In this case the question does not ask if a set of nodes is activated at the same time but asks if they can be deactivated which is not the same thing. In this case the ASEP cannot be applied because the questions cannot be seen as a satisfaction of a conjunction.

Instead of using the conjunction of each node, we can use the same property with the conjunction of the negation of each node. By looking the problem that way, we can then use the ASEP with other **OrToOpt** constraints. Implementing

this solution would imply adding new conditions to the propagation methods in **DecisionNode**. This new condition would then increase the complexity of the algorithm structure and a refactoring would probably be needed in some parts of the decision tree module.

8.4 Summary

In this chapter we analyzed and interpreted results coming from measures taken by the new version of the mutation testing tool. We also managed to fix errors that were found earlier in the previous version and the new questioning algorithm using decision tree to prune questions showed encouraging results being able to prune up to 50% of the initial question set.

In the last part of this chapter we also summarize some improvements that could be made to the new tool. We proposed to introduce cross tree constraints and to enrich the proposition in term of mutations. On a more practical point of view, we can also create a xml parser to link the featureIDE model with the encode model to get rid of the current encoding format and we discussed the possibility to improve the newly created questioning algorithm.

Chapter 9

Conclusion

The goal of this master thesis was to continue the work started by A. Deckers in his own master thesis. His work defined a strong theoretical approach of how mutation testing would be used in this particular type of model as well as a first version of a mutation testing program for CFM models [6]. This program was designed to take a CFM model encoded by the user as input, then from the model, a couple of changes called mutations were made. With those mutations we can then generate a series of questions whose answers from the model designer help us to identify some potential design flaws.

We first started by trying the old version of the tool in order to see if it seemed ready to be used by a group of students. To do so we added new error messages in order to know why the program crashed and where is the problem if an erroneous input was introduced. After giving this program to the class of students, we asked them to fill a survey regarding their use of the tool in order to have a feedback. This feedback helped us to identified different test cases that caused abnormal behaviors of the tool (i.e. no questions showed). The idea was first to fix those problem and then see the effect of new mutation operators but we were stopped by the difficulty to understand the code base as it contained bad smells that would make any modifications dangerous for the program's correctness. Furthermore, the program used small structures called connected-pairs used for the mutations that were poorly defined.

The solution was then to proceed in a complete re-implementation of the testing tool. This re-implementation contains a static visitor design pattern used for the mutation module of the new program. We also redefined the connected-pair generation and now they are called sub-models. The new qub-model generation algorithm makes sub-models smaller without losing their utility which is ensuring that mutations are done in a way that they actual infer changes in the model.

After implementing the new version of the testing tool we decided to use properties of boolean conjunctions in order to improve the generation of questions. This new methodology uses a decision tree algorithm that allows the program to guess the answer to questions that are not asked yet based on the answer to previous questions.

In the end, our contribution to the creation of a mutation testing tool shows encouraging results. The new version of the tool fixed all the bugs that were reported by the survey. The decision tree strategy for the questioning steps has proven itself useful in different cases as well and many new possible contributions to this subject can still be explored.

Appendix A

Form results

Lab1 (extended) feedback

8 réponses

Publier les données analytiques

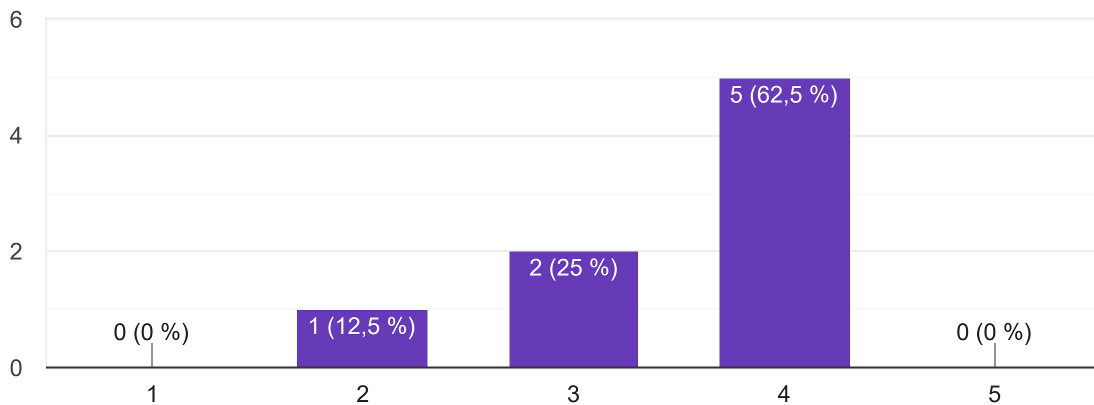
Please upload your 3 .txt files used as input to the testing tool.

8 réponses

Format.

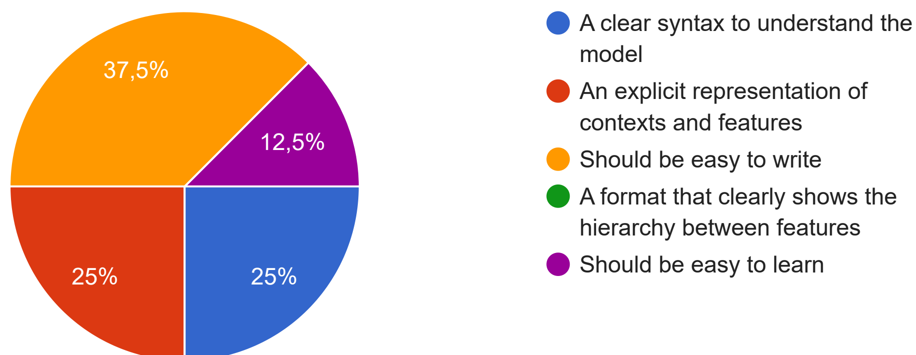
How hard was it to write your input in this format?

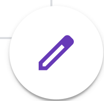
8 réponses



For you, what is/are the most important aspects that the input of the testing tool should have?

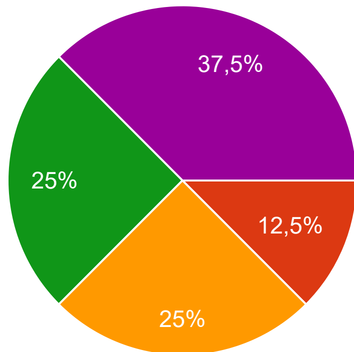
8 réponses





Which of these aspects were sufficiently supported by the current format?

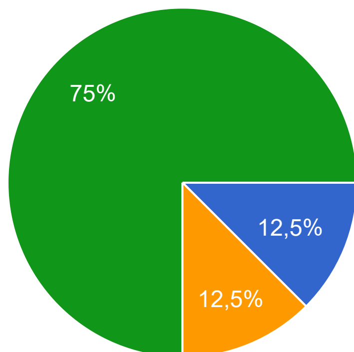
8 réponses



- A clear syntax to understand the model
- An explicit representation of contexts and features
- Should be easy to write
- A format that clearly shows the hierarchy between features
- Should be easy to learn

Do you think it is better to have a grammar-check (underlining) for highlighting errors while writing your input to the tool, or rather a clear and precise stack trace in case of errors after launching the tool?

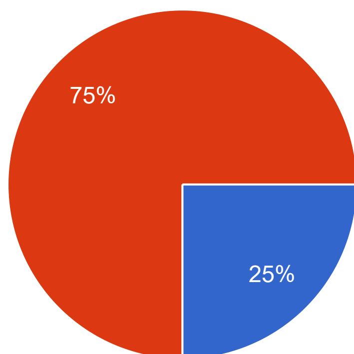
8 réponses



- I strongly prefer a grammar-check over using the stack trace
- I'd rather prefer the grammar check over the stack trace
- I think both are equally useful
- I'd rather prefer the stack trace over the grammar-check
- I strongly prefer the stack trace over using the grammar-check

Would you be willing to install and understand a new tool taking feature IDE's models as input as opposed to spending more time on writing the input yourself?

8 réponses



- More time on modeling without any additional tool
- More time to install a tool but a faster modeling step

Optional : What suggestion(s) do you have to improve the current input format?

2 réponses

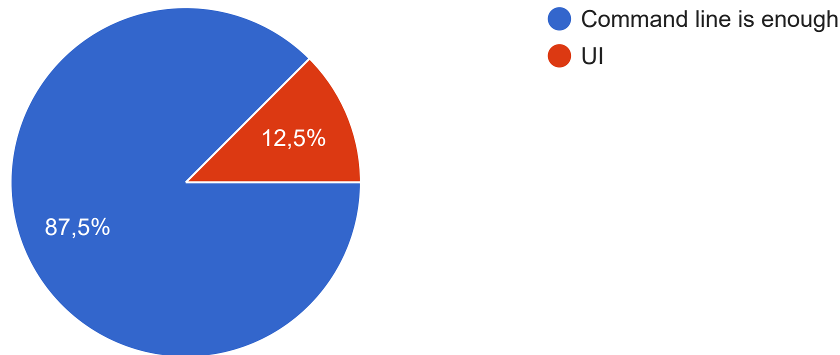
Allow empty lines in text files for greater clarity.

Alloy white lines to visualise "blocks" of features

Tool

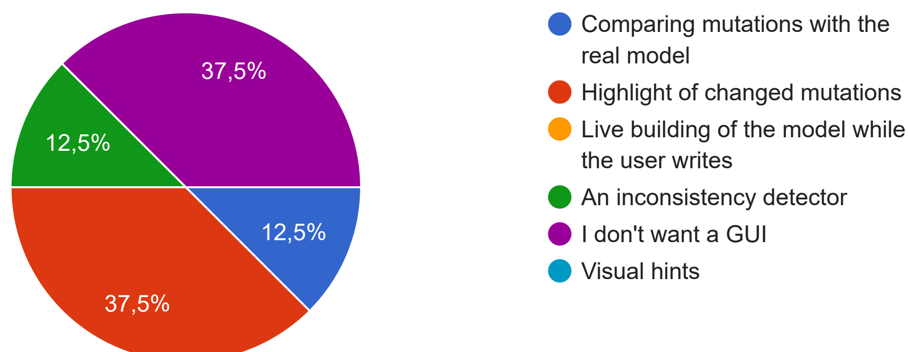
Do you think the command line output format is enough or do you think the output could be improved with a UI?

8 réponses



If you prefer a graphic interface what features do you think are important?

8 réponses



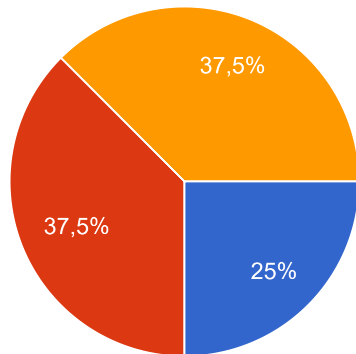
If you chose "Visual hints", please indicate what kind of hint you would find useful

0 réponse

Il n'y a actuellement aucune réponse à cette question.

Instead of a generic list of questions, would you prefer being able to specify a list of features/contexts/mapping relationships and only get recommendations or questions about them ?

8 réponses



- I strongly prefer a generic list of questions over getting recommendations about a list…
- I'd rather prefer a generic list of questions over getting recom…
- I think both are equally good
- I'd rather getting recommendations about a list…
- I strongly prefer getting recommendations about a list…

Optional : What suggestion do you have to improve the output of the tool?

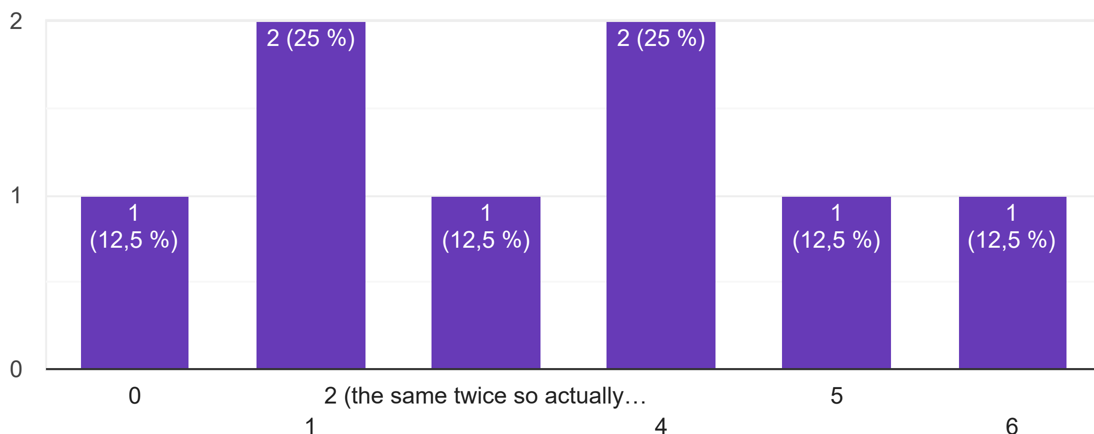
Une réponse

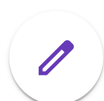
In our case, we had 2 contexts that were activating the same thing and the tool did not detect. So we received no recommendation on what we did but there was repetition in our contexts

Results

How many questions were you asked? (between 0 and 100)

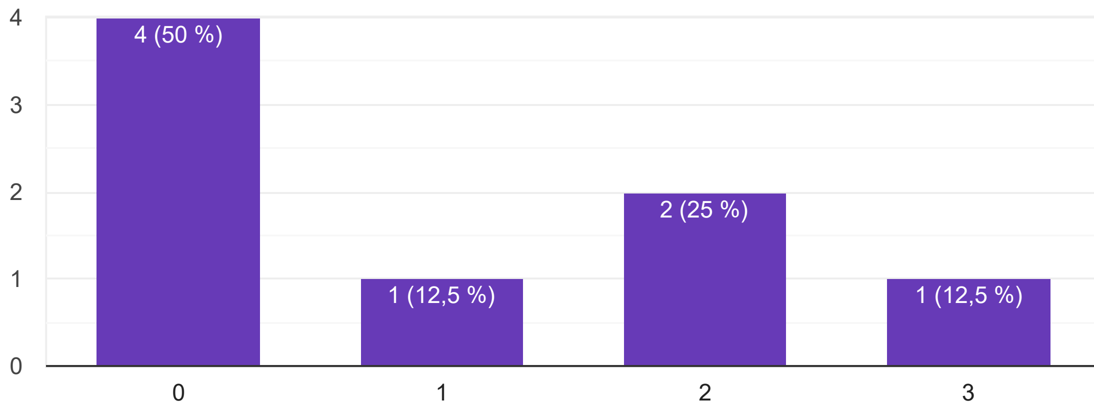
8 réponses





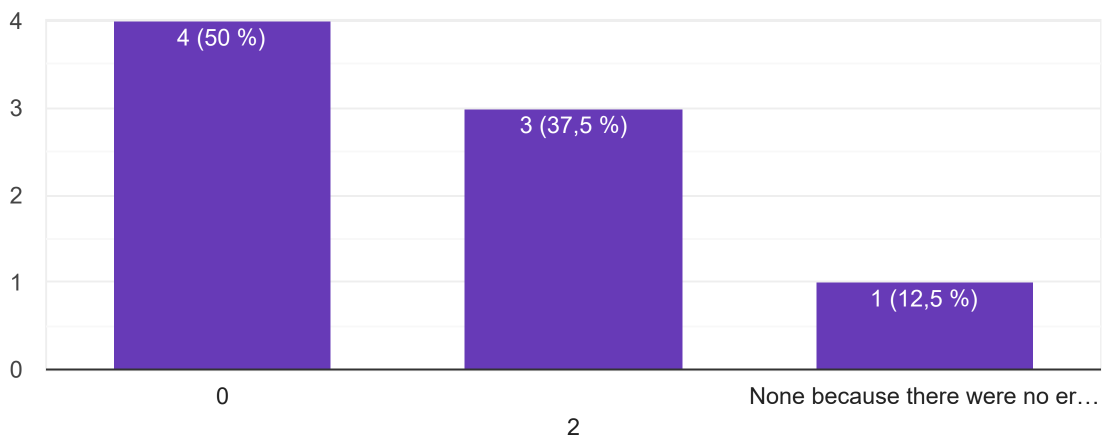
How many recommendations did you receive? (between 0 and 100)

8 réponses



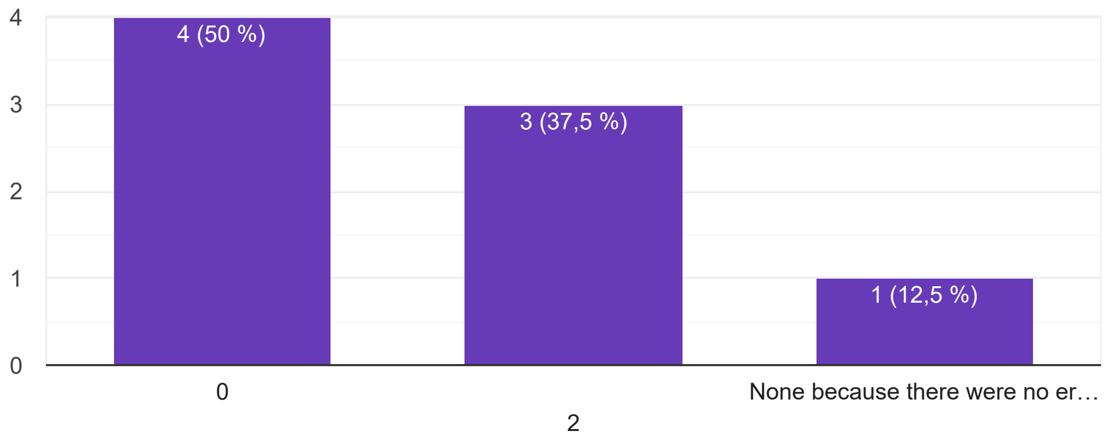
How many of these recommendations did you really follow?

8 réponses



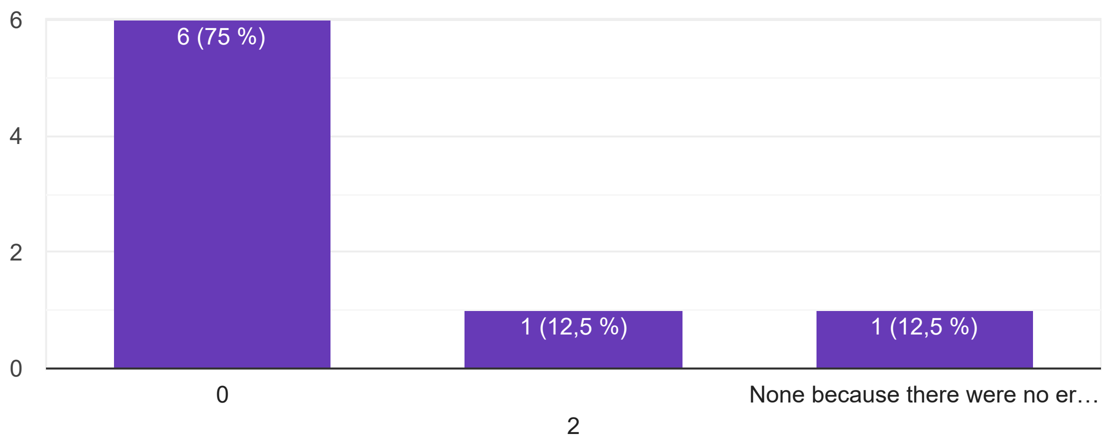
Among the recommendations you received, how many of them did you apply to the letter?

8 réponses



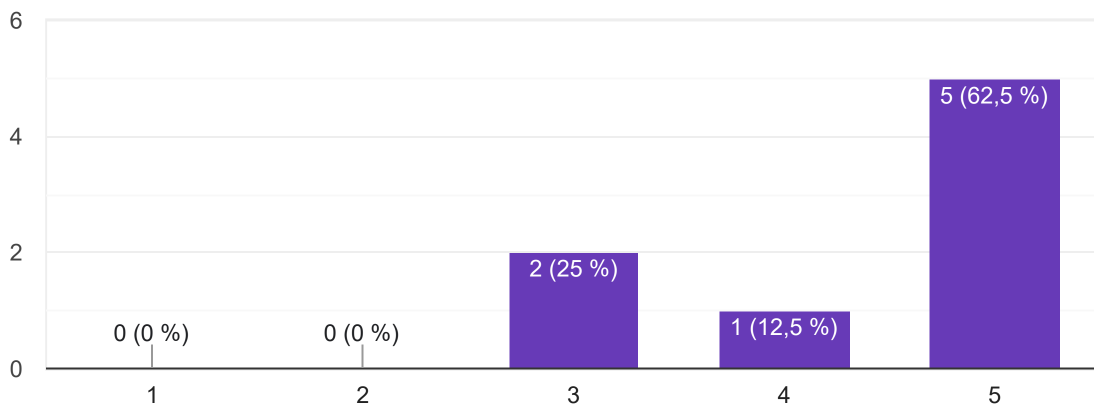
Among the recommendations you received, how many of them helped you to find errors in the model?

8 réponses



Did you find that the questions asked by the tool were easy to understand?

8 réponses



Optional : What suggestion do you have to improve the questions and the recommendations provided by the tool?

0 réponse

Il n'y a actuellement aucune réponse à cette question.

Bug report : please describe here any encountered bugs while using the tool.

4 réponses

A got twice the same question then it was over (our context model is small maybe this was the reason for juste "one" question)

the tool recommended checking features but the feature name was missing

Some questions appear twice

No question and no recommandation received

Metaquestion : Do you have any suggestion or remark to improve this questionnaire?

0 réponse

Il n'y a actuellement aucune réponse à cette question.

Ce contenu n'est ni rédigé, ni cautionné par Google. Signaler un cas d'utilisation abusive - Conditions d'utilisation - Règles de confidentialité

Appendix B

Class diagram

MapReader
mappingFile : String
contextTree : Tree
featureTree : Tree
readFile() : MappingModel
processLine()

MappingModel
dict : Dictionary
dict : Dictionary
size : int
getSize() : int
getKeys() : List
isEmpty() : bool
addEntry()
getEntry() : Set
copy() : MappingModel

CFReader
filename : String
fileType : int
readFile() : Tree
processLine()

SubModelsGenerator
featModel : Tree
ctxModel : Tree
mapModel : MappingModel
subModelList : List
genSubModelsList() :

Node
name : String
parent : Node
children : List
isRoot : bool
getAllChildrenName() : List

Constraint
nodeSet : Set
isEmpty() : bool

OrConstraint

...

Tree
root : Node
size : int
addConstraintGroup()
removeNode() : Node
get() : Node
getConstraintGroup() : Constraint
getParent() : Node
copy() : Tree

DecisionNode
depth: int
questionMask: Dictionary
depthList: List
predictedQuestions: OrderedDict
questionDict: Dictionary
questionToAsk: String
left: DecisionNode
right: DecisionNode
propagateNo()
propagateYes()

SubModel
ctxModelList : List
mapModel : MappingModel
featModel : Tree
copy() : SubModel
accept()

Main

Mutation
is_applicable() : bool
visitSubModel() : Question
visitSubModel() : Question

OrToAlt

...

Question
id : int
question : String
answer : List
mutant : SubModel
appliedMutation : Mutation
ctxNameList : List
featNameList : List

QuestionStrategies
questionStore : QuestionStore
recommendataionList : List
questionDict : Dictionary
applyAskingStrategy()
genRecommendation()
applyAnswer()

Recommendation
ctxNameList : List
featNameList : List
mutation : Mutation
showRecomMessages() : String
printNameList() : String

QuestionStore
store : Set
add()

DecisionTree
questionList : List
root : DecisionNode
nquestions : List

...

parent child Use

Appendix C

New mutation implementation

OptToMan

```
class OptToMan(Mutation):

    def __str__(self):
        """
            Pre :
                /
            Post :
                Returns the string representation of the mutation (i.e. the name of the mutation)

        """

        return "Opt-To-Man"

    def is_applicable(subModel,ctxConstraint, featConstraint):
        """
        When there is a mand on the optional and a context in the feature ->ctx
        """
        return isinstance(ctxConstraint,OptionalConstraint)

    def visitSubModel(subModel,ctxConstraint, featConstraint):
        newSubModel = subModel.copy()
        ctxModel = newSubModel.findCtxModelFromConstraint(ctxConstraint)
        targetConstraint = MandatoryConstraint()
        newConstraint = Mutation.applyRegularMutation(ctxModel,ctxConstraint, targetConstraint)
        nodeNameList = [node.name  for node in newConstraint.nodeSet]
        return Question("Does {} have to be activated in any configuration?".format(nodeNameList[0]),["yes","y"] ,newSubModel,OptToMan(),ctxNameList = nodeNameList)
```

Bibliography

[1] Benoît Duhoux. *L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte*. PhD thesis, Université Catholique de Louvain-la-Neuve, 2015-2016.

[2] 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*, pages 15–22, 2018.

[3] Benoît Duhoux, Bruno Dumas, Kim Mens, and Hoo Sing Leung. A context and feature visualisation tool for a feature-based context-oriented programming language. In *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE 2019*. CEUR-WS, 2019.

[4] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. Dynamic visualisation of features and contexts for context-oriented programmers. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2019.

[5] Pierre Martou, Kim Mens, Benoît Duhoux, and Axel Legay. Test scenario generation for feature-based context-oriented software systems. *Journal of Systems and Software*, 197:111570, 2023.

[6] Pierre Martou Audric Deckers, Kim Mens. Testing the design of context-oriented software through mutation testing. Master's thesis, Université Catholique de Louvain, Louvain-La-Neuve, BE, June 2023. Available at `https://dial.uclouvain.be/memoire/ucl/object/thesis:40644`.

[7] Henry Lieberman and Ted Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM systems journal*, 39(3.4):617–632, 2000.

[8] Kim Mens. Lecture notes in software maintenance and evolution, chap 10: Contextorientedprogramming p22., October 2022.

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

[10] Refactoring guru. Strategy, 2024.

[11] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda). *Software Engineering Institute, Carnegie Mellon University technical report*, 1990.

[12] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In *Modeling and Using Context: 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007. Proceedings 6*, pages 178–191. Springer, 2007.

[13] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[14] S. Misra. Evaluating four white-box test coverage methodologies. In *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*, volume 3, pages 1739–1742 vol.3, 2003.

[15] Charles Pêcheur. Lecture nots in sofware quality assurrance, chap 5: More testing, p.65., march 2023.

[16] 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)*, pages 1–10. IEEE, 2015.

[17] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-based product-line testing: Effective sample generation based on feature-diagram mutation. In *Proceedings of the 19th International Conference on Software Product Line*, pages 131–140, 2015.

[18] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[19] Kim Mens. Lecture notes in software maintenance and evolution, chap 4: Software maintenance & evolution, bad code smell, p.13., Novemeber 2022.

[20] Source Making. Long method, 2024.

[21] Ihechikara Vincent Abba. Single responsibility principle, 2022.

[22] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *2009 ieee 31st international conference on software engineering*, pages 611–614. IEEE, 2009.

[23] Chico Sundermann, Kevin Feichtinger, José A Galindo, David Benavides, Rick Rabiser, Sebastian Krieter, and Thomas Thüm. Tutorial on the universal variability language. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*, pages 260–260, 2022.

[24] Refacroring guru. Visitor, 2024.

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