UCLouvain

École polytechnique de Louvain

Context-Specific Composition of Features in Context-Oriented Programming

Author: **Pierre-Olivier Martin**
Supervisors: **Kim Mens, Benoît Duhoux**
Reader: **Charles Pecheur**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Summary

Abstract

This master's thesis takes place in research about Context-Oriented Programming in an experimental language, the Feature-Based Context-Oriented Programming Language.

The objective is to enhance the relations between the contexts and the features in this language.

The Feature-Based Context-Oriented Programming Language used a context model, a feature model and a mapping which represents the relations between the two models.

This master's thesis shows that it is possible to improve the mapping and the models.

First, the mapping declaration is more flexible and expressive thanks to the introduction of the OR and NOT operators (in addition to AND) and the possibility to reuse and split an expression in sub-expressions. Together, it makes possible to use the mapping declaration in large and complex applications.

Second, the context and feature models are merged with the mapping to form a new one. This entire model is optimized, and its time complexity is decoupled from its size.

Third, the performance of the mapping is increased thank to the use of gates. In addition, it has been shown that, no matter how the mapping is declared, its average performance remains the same.

Finally, the introduction of the notion of transaction and of the different strategies are used to perform the context changes.

A new implementation of the Adaptations mechanism is also proposed based on pointers.

This improves the performance of the language and the speed of the behavior changes. In addition, this new version is compliant with multithreading whereas the original version was not.

This works show that it should be possible to make an implementation of FBCOP usable in any large and complex application.

Acknowledgments

I really want to thank my supervisors, Prof. Kim Mens and Benoît Duhoux, who guided me throughout this master's thesis and without whom this work would not have been possible.

I also want to thanks Prof. Charles Pecheur who accepted to be my reader and for his time spends to read this master's thesis.

I also thanks all the professors and assistants who taught me during my studies. Without the acquired knowledge, none of this would be possible.

I finally thanks my parents and my sister who supported me during all my studies.

Chapter 1

Introduction

This master's thesis takes place in research about Context-Oriented Programming in an experimental language, the Feature-Based Context-Oriented Programming Language.

The vocation of this master's thesis is to enhance the relations between the contexts and the features in the Feature-Based Context-Oriented Programming Language.

The Feature-Based Context-Oriented Programming Language make the distinction between the features, which represents a behavior of the system, and the contexts, which represent any data that can influence the behavior of the system. The relations between these two kinds of entities model how the system must adapt according to the situation. These relations are a critical part of this language as they can restrain or extend the adaptability of the system which uses it.

After a reminder of the background knowledge needed for this master thesis, the approach used is described and the related work is reviewed. The implementation of the new model is then described as well as the transactions used in this model. The work presented is finally discussed and validated.

Secondly, another language improvement is proposed even if it is not directly related to the initial subject of this master's thesis.

Finally, some possible subjects of research are proposed.

Chapter 2

Background

This chapter introduces the concepts necessary for the understanding of this thesis. Firstly, a brief overview of the Context-Oriented Programming (COP) paradigm will be given. The second section will be about how to model such programs. The last section of this chapter will describe the Feature-Based Context-Oriented Programming, a particular class of Context-Oriented Programming languages.

2.1 Context-Oriented Programming

The paradigm of context-oriented programming was created to address the problem of context-dependent behavior in classical programming techniques.

Nowadays, with the rise of smartphones and the internet of things, more and more applications are dependent on their contexts and want to adapt their behavior accordingly. But classical paradigms like functional or object-oriented programming are not designed for this. The programmer thus ends up building an overly complex system to handle this flexibility for each software system he makes.

The context-oriented paradigm integrates the notion of context-dependent behavior on top of object-oriented programming [12, 24]. It offers the programmer the ability to implement more easily a context-dependent application without writing an entire dynamic system from scratch.

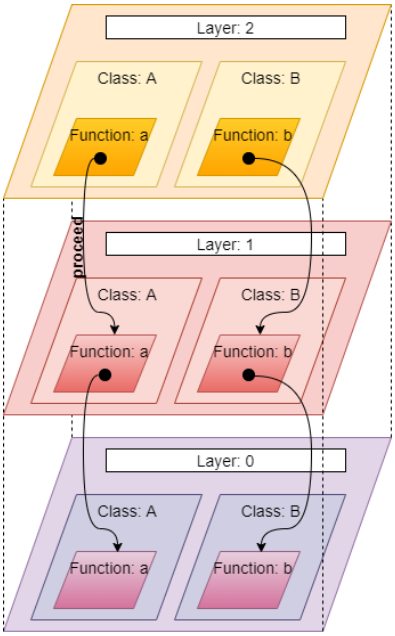


Figure 2.1: Layers of functionality in context-oriented programming

To do so, this paradigm introduces the notion of layers (illustrated in figure 2.1). A *layer* is a group of *behavioral variations* specific to a particular context. It means that a piece of code (the behavioral variation) will be dynamically deployed during the execution when the associated context becomes active. As different layers can be active at the same time, the paradigm needs a mechanism to call other layers. The *proceed* mechanism is an answer to this need. When the proceed is called in a function, it calls a function of the same

name in a previously deployed layer (as depicted in figure 2.1). It is comparable to the use of a *super* call in object-oriented languages to call the parent function. The main difference is that the result of a proceed call depends on the stack of layers, which can change during the execution, whereas the super call is statically linked to the parent chain. The dynamicity of the proceed call is due to the fact that, when a layer is deployed, it is loaded on the top of the stack. The functions deployed last are thus called first and the older ones can be reached thanks to the proceed statement. When a layer gets deactivated, it is simply removed from the stack (no matter where it is in the stack), so that its behavior is no longer visible.

Different languages implement this paradigm such as ContextL [3], ContextS [12] or ContextJ [13] respectively based on Lisp, Smalltalk and Java. In the above, only the "layer-based" context-oriented programming languages were explained. Other context-oriented programming languages [24], relying on different building blocks, exist such as for example SubjectiveC [8], ContextTrait [10], Ambience [9], PhenomenalGem [23] and so on. Nevertheless, they remain quite similar in spirit to the layered-based context-oriented languages explained above, so the mechanism underlying these other languages won't be detailed here.

2.2 Feature Modeling

In most COP languages, there is no notion of feature modeling to model the different functionalities or their behavioral variations. The language used in this master's thesis (presented in the next section 2.3) uses a more complex approach based on feature modeling. This section therefore introduces the basis of this modeling technique.

2.2.1 Feature Diagram

The feature diagram models the features of a software system [16]. The software is represented as a tree structure where each node is a feature. A feature is a part of the software which represents a specific behavior. To model the interactions of the features, different constraints are used between a parent node and its children. These constraints are the optional children (Optional in figure 2.2), always active children (Mandatory in figure 2.2), one and only one active children (Alternative in figure 2.2) and at least one active children (Or in figure 2.2). According to these relations, the software can have different configurations. Each feature can be active or inactive. A configuration is a specific state of the feature diagram where each feature is either active or inactive. The configuration is valid if it respects all the constraints.

An example of Feature Diagram is given in figure 2.2. This example is a

simplified version of the car factory line given by Hartman [11]. In this example, the maps feature is not needed by the software to work but if it is active, it needs at least a road map to work like the European or the American one. The system must also provide an interface which can be either a Card-Slot or an USB.

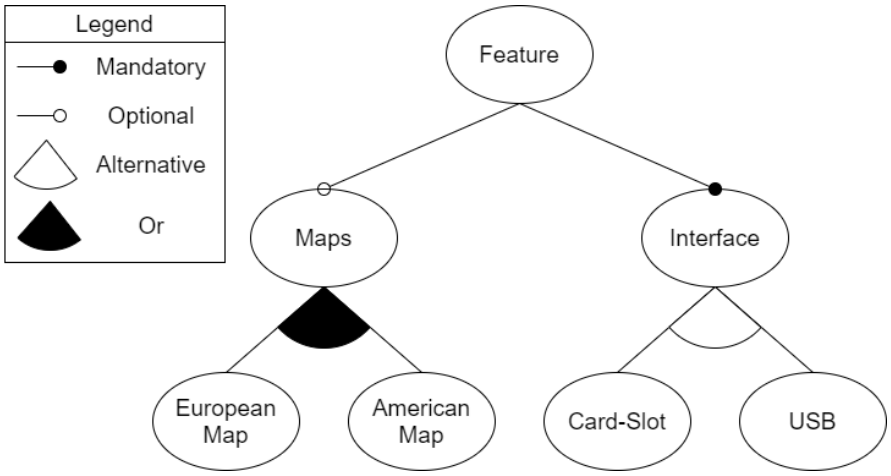


Figure 2.2: Example of Feature Diagram inspire from [11]

2.2.2 Multi Product Line Feature Model

Linking feature modeling to context-aware system development, the Multi Product Line feature model (MPL Feature Model) was introduced to handle the variability of features in production lines with different contexts [11, 1, 19]. It allows to only draw one model for several product lines and not make several ones with minor changes, depending on the context in which the product line will be deployed.

To reach this objective, the model is composed of three parts: the *context variability model*, the *feature model* and the *relations* between them. The context variability model represents the different possible contexts of the product line. In a similar way, the feature model groups all the possible features that the product line can have which may depend on the context.

The context variability and feature models are both represented as a Feature Diagrams. For a given context configuration, the relations determine a specific configuration of contexts and features for the product line. This resulting configuration is valid if it respects all the constraints of the model. The context variability and feature models are related through relations which are implications between the contexts and the features. For example, on figure 2.3, if the product line is in Europe, the European Map will be integrated by default in the system.

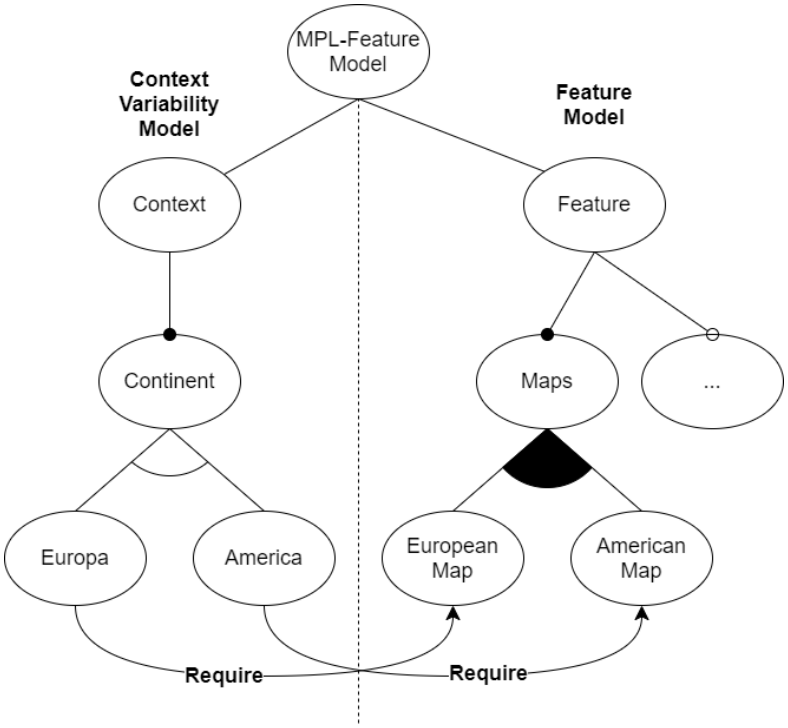


Figure 2.3: Example of Multi-Product Line Feature Model inspire from [11]

2.3 Feature-Based Context-Oriented Programming

Classical COP language implementations make no explicit distinction between contexts, features and behavioral variations, in the sense that all these parts are grouped in a same concept, the layer[1]. The Feature-Based Context-Oriented Programming (FBCOP for short) approach tries to make this distinction more clearly in order to achieve better modularity and more expressiveness [7]. The Context-Oriented software architecture is inspired by [20]. But FBCOP replaces the notion of layer by a more advanced model that is inspired by the MPL Feature Model explained in the previous section 2.2.2 as suggested by Hartmann and Trew [11]. The Feature-Based Context-Oriented Programming approach builds on the following concepts, summarized in figure 2.4.

- **Context** : A context represents any situation considered as useful by the designer to distinguish different behavioral adaptations.

[1]The same remark holds for most non-layer-based COP language implementations as well.

- **Behavioral variation** : A behavioral variation is a module of code that can be used to dynamically alter the behavior of existing classes or previously installed behavioral variations.
- **Adaptation** : An adaptation groups a behavioral variation and a class it alters. A same behavioral variation can be used in several adaptations if it is used to alter several classes. An adaptation can be activated or deactivated. When it is activated, the behavior is added to the class. When it is deactivated, the class in no longer adapted with that behavior.
- **Context Model** : The context model is similar to the context variability model of the previous section 2.2.2. It is used to model all the situations for which the system can adapt its behaviors.
- **Feature Model** : The Feature model is equivalent to the one explained in section 2.2.2. It models all the possible behaviors that the system can have. Most of the nodes (called features) are composed of a behavioral variation in one or several classes. The other nodes (called abstract features) have no behavior by itself but help to organize the tree. The features, by making the link between the behavioral variation and the classes, create the adaptations. These adaptations will be activated and deactivated with the feature which created it.
- **Mapping**: The Mapping is composed of the relations between the context model and the feature model. These relation will propagate the context changes to the feature model and activate or deactivate the features according to the relations.

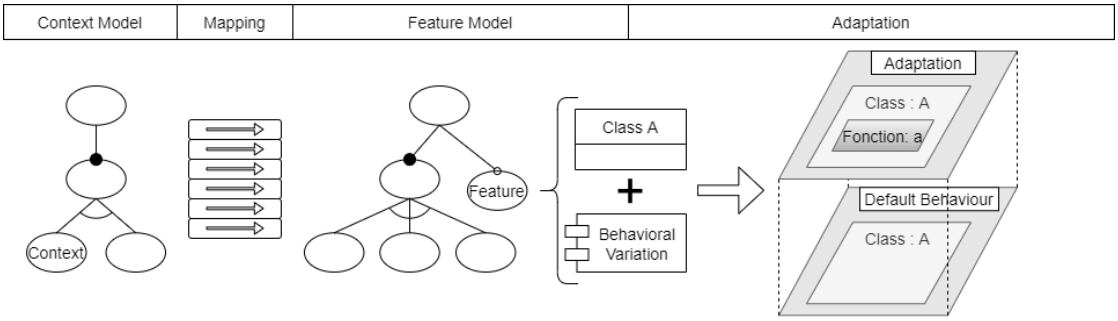


Figure 2.4: FBCOP Modeling

During system execution, the user, the sensors or the program itself can activate or deactivate the contexts. When a context becomes active or inactive the mapping

is notified. According to the mapping relations, the mapping will activate or deactivate the features required by the actual contexts. When a feature becomes active/inactive, it activates/deactivates its adaptations to adapt the behavior of the system according to the contexts.

For example, suppose building the system presented in the MPL feature model section with some adaptations for a GPS. The GPS must load the correct map according to the continent where it is. The model is presented in figure 2.5. The difference with the model presented for the car factory is that only one map can be active in the feature model (the Or becomes an Alternative). Indeed, even thought the GPS always know all the maps, it only uses one at a time.

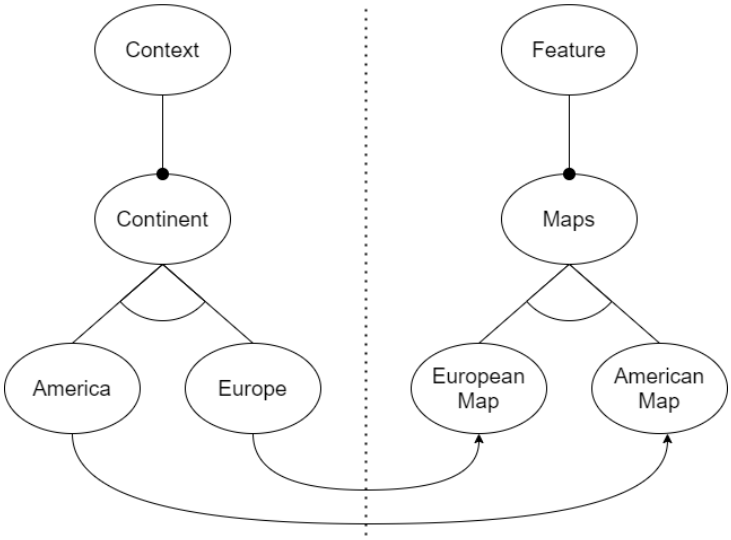


Figure 2.5: GPS map model

When the GPS starts, a context will be activated by default according to the specification given by the developer, let's suppose the *Europa context* in our case. The mapping will then activate the *European map feature*. When the GPS will use the map, it will use the European one. If the car is put on a plane and arrives in America, a sensor will detect it thanks to the GPS position and change the context. More precisely, it will deactivate the *Europa context* and activate the *America context*. The mapping will then, according to the relations (see figure 2.5), deactivate the *European Map* and activate the *American Map*. At this time, the GPS will use the American map for its operations.

The FBCOP implementation is build on top of the Ruby language.

2.4 Conclusion

In this master's thesis, the Feature-Based Context-Oriented Programming language will be used. This language adapts its behavior according to the context as the other COP languages but also increases the modularity of its applications and the expressivity of its contexts and features representation. To achieve this result, FBCOP uses its own model which is inspired from the Multi-Product Line Feature Model. The MPLFM models and links together the contexts and the features of a system.

Chapter 3

Approach

The mapping of contexts to features is a critical part of the language. By modeling explicitly the links between the contexts and the features, it becomes easier to understand how the system adapts its behavior with features when the contexts change. To introduce this chapter, an overview of the current mapping in FBCOP is given. Then the motivations to innovate this mapping are described. Afterwards, a review of other possible mappings is discussed. At the end of this chapter, a new version of the mapping is presented, thus contributing to the main part of this master's thesis.

3.1 Current FBCOP mapping

The current context-feature mapping [7] (example in Listing 3.1) is composed of a set of ordered rules (the rule order is top to bottom). Each rule is composed of two parts, the contexts expression (the array to the left of the arrow) and the features it implies (the array to the right of the arrow). The rule is triggered if all the contexts inside the context expression are activated (Conjunction). When the rule is triggered, it activates the features from the left to the right. As can be seen on Listing 3.1, different rules can activate a same feature. This allows mimicking a disjunction as a feature can be triggered if either one or another condition is triggered.

In a more logical form, this kind of mapping can be expressed as the following statements where the order is simply left to right.

$$
\begin{aligned}
&\mathit{mapping} ::= \mathit{expression} \mid \mathit{mapping} \wedge \mathit{expression} \\
&\mathit{expression} ::= \mathit{contexts} \Rightarrow \mathit{features} \\
&\mathit{contexts} ::= \mathit{context} \mid \mathit{contexts} \wedge \mathit{context} \\
&\mathit{features} ::= \mathit{feature} \mid \mathit{features} \wedge \mathit{feature}
\end{aligned}
$$

The order of the mapping is very important as it defines the order in which the features will be activated and so the order in which the adaptations will be deployed. If two features refine a same functionality, activating them in a different order thus may have a different effect on the system's behavior. Deactivations are performed in the reverse order and always before the activations if both are required by the context changes. To illustrate the impact of the adaptations order on the system's behavior, let's imagine that the clear and rainy weathers are activated at

the same time following the mapping describe on the Listing 3.1[1]. The result will be that the hard break is activated first and then the soft break. The behavior of the car will be to break softly as it is the last deployed function. The hard break will not be used because it was deployed before the soft break (the break features do not make a proceed).

Listing 3.1: Mapping of car breaks: The car can break hard on a dry road but need to break softly on a wet or snowy road.

```
class CarMappingDeclaration < MappingDeclaration
  include Singleton
  def initialize()
    @mapping = {
      [
        CarContextDeclaration.instance.clear_weather_context()
      ] => [
        CarFeatureDeclaration.instance.hard_break_feature()
      ], [
        CarContextDeclaration.instance.rainy_weather_context()
      ] => [
        CarFeatureDeclaration.instance.soft_break_feature()
      ], [
        CarContextDeclaration.instance.snowy_weather_context()
      ] => [
        CarFeatureDeclaration.instance.soft_break_feature()
      ]
    }
  end
end
```

3.2 Motivation

The actual mapping offers a simple interface for the developer but becomes difficult or even impossible to use in large applications or when more complex mappings are needed. Let us explain why.

The first problem is that even a simple mapping can become a large expression. Let's revisit the mapping example of the previous Listing 3.1. In a formal form, it can be expressed as $(clear\_wheather \Rightarrow hard\_break) \wedge (rainy\_weather \vee snowy\_weather \Rightarrow soft\_break)$. Whereas formally only this expression is needed, the only way to express the second rule is by splitting it in two. For a small

[1] A smart developer should constraint the weather contexts and the break features with an Alternative to avoid this case.

mapping, this is not a real problem, but when the number of features becomes large, this impossibility to express simple expressions with a simple rule makes the mapping even larger.

A second problem is when the developer wants to express the negation of a context. Of course in most cases, he can just define or uses other contexts to express it. For example, to express the negation of the clear weather in the previous example, the snowy or rainy weather can be used . But in some cases it is not so simple. As example, let's take which kinds of passengers are in the car as context. The passengers can be a baby or an adult or a child which are all context children of the context passenger with an Or constraint. In this case, it is impossible to express they are no baby on board, as knowing they are an adult or a child give no information about the presence of a baby. In fact, it is impossible to express it with that contexts. The only way is to modify the contexts and add a mirror of the passenger contexts with the ones who are not in board. But even if it is possible to express the negation of a context, the developer will not want to make the context model and/or the mapping more complex to express a simple relation.

The actual mapping is thus limited by its expressiveness. This master thesis has the vocation to search and implement an alternative to the actual mapping to solve these problems.

3.3 Related Work

Before trying to enhance the actual mapping, existing mappings used for linking models must be explored and reviewed in order to get a more general picture of the problem.

In the most common layer-based COP languages [13, 12, 4], the mapping is implicit and groups a single context with one layer by using the same name for the both. A one-to-one relation is also used in context-feature modeling [11, 1]. As it is not based on using the same name, a single context can activate several features. However activate a single feature with several contexts is impossible.

Two other COP languages introduce each a new concept to a mapping using one-to-one relations. SubjectiveC [8] introduce the possibility to use the complement of a context (its negation). Even if it is considered as syntactic sugar, this addition not only makes the mapping and model easier to use but adds expressiveness to the mapping. On the other side, Ambience[9] allows using a combination of contexts to activate a behavior. More exactly when several contexts are involved in the relation, the language creates a new context that represents the combination and links this new context to the layers. This context combination allows declaring a many-to-one relation in an architecture that only implement the one-to-one relation between context and behavior.

EventCJ [15] is a COP language where the transition of layers (activation, deactivation and replacement) is based on events. The events are emitted by the call of some functions. These events can be global and sent to all instances or specific to one or several objects. When an object gets an event, it selects the layers according to the transition rules it has. These rules are quite basic. On a specific event the object can activate, deactivate or replace a layer. As the events are punctual, combinations are impossible.

A totally different kind of mapping was also proposed for COP languages with the use of ontology based mapping[22, 14]. But this approach makes the mapping a lot more complex due to its different kinds of rules between the models.

Other context-aware programming approaches also deals with context-feature mapping. Different approaches are commonly used [21]: Key-Value, logic, rule-based, graphical or ontology based mapping. These mapping offer at least a full propositional logic support with different representations except for the key-value mapping that only allows for very simple expressions. These representations often involve the operations of comparison as contexts can be of any types whereas FBCOP only uses boolean. Two specific context-aware languages need to be reviewed deeper. The first is from Jacopo [18]. His definition of the mapping is logic-based but allows using contexts and features on the two sides of the relation. So a feature can be used in the condition and a context can be activated/deactivated from a condition. The second is from Trullemans [25]. This mapping is based on a rule-based system (IF condition THEN action). The difference is that the rules are split in three levels, the basic block (the context and action), the template (the operations like AND, OR or more complex ones like "user X is in a room Y") and the rules which are based on filling the template with contexts to make the condition. The purpose is that each level requires different level of knowledge. Thanks to these abstractions, a user without any particular background can define the rules for a system easily by using existing blocks. A more advanced user can define new templates and a developer can do what he wants. For example, a developer creates the blocks needed to communicate between an arduino and a led strip or with a presence sensor. Someone else creates a pattern which models if someone is in a room. So a basic user that bay a led strip , an arduino and a sensor will be able to turn on the leds only when someone is in the room without knowing anything of computer programming. This mapping shows that anyone can make a complex mapping with the right level of abstraction.

3.4 Proposal

The following proposal is the solution proposed by this master's thesis to answer the problem of the expressivity of the current mapping. Firstly, the new mapping

is presented. Then, two smaller changes are proposed for the context and feature models.

Mapping

In the case of FBCOP, the mapping must join two trees (example on the figure 2.5) where each context or feature can be active or inactive, and in consequence considered as boolean. A single rule is composed of two elements. The antecedent that models the condition (left part) and the consequent that models which elements are dependent on that condition (right part).

$$antecedent \Rightarrow consequent$$

The arrow between the antecedent and the consequent models the relation "if the antecedent is matched then the consequent is applied". In logic it is equivalent to an implication.

The first question to ask is : *At which place feature or context can be used?* The current mapping allows using context only in the antecedent and feature only in the consequent. As seen in the related work [18], this is not always true. But allowing more flexibility here could create undesirable behaviors. The first problem would be an oscillating rule as with the implication $\neg A \Rightarrow A$. The condition will never be in a permanent state as the consequence changes the state of it. The second problem will be a non-defined state as with the implication $B \Rightarrow B$. In this case what should be considered ? Must the state of B be 0 or 1? Is it 0 until B is activated by another rule and stay 1 until the end ? Moreover, it will not change the expressiveness of the mapping if these loops are banished. Indeed, with no loop, all features must depend on an antecedent. So to use a feature in the left part of the implication, the antecedent of this feature can be used instead. For the use of a context in a consequent, it means to create a context dependent on other contexts, which can be seen as a reasoning about contexts. And as for the feature, this "reasoning" can be replaced by its antecedent directly where it is used. To avoid these unnecessary problems, the actual constraints can be kept. The contexts must only be used in the antecedent and the features in the consequent.

The second question is: *Which operators can be used in the implication?* In the case of the condition, full propositional logic can be used without any problem. If the current version only accepts the AND, the new version must give the opportunity to use the AND but also the OR and the NOT. With these three operations, any boolean condition can be express in the mapping. To allow the developer to express the antecedent in its shorter or more understandable form, no constraint about using these operators will be made. For the consequent, allow the same operators inside of it is more complex. The current version of the consequent only uses AND

between the features. Introducing an OR is meaningless. It is equivalent to giving an order to someone by saying "You must do that or that". With this order, what must the man do? The first order, the second one or both? The second statement that could be introduced is the NOT. If a positive implication is the activation of the feature, the opposite could be considered as its deactivation. But some problems arise. The first is what happens when the condition is not matched? The feature must be activated or not? But as in logic $a \Rightarrow \neg B$ is not equivalent to $\neg a \Rightarrow B$, the answer should be that the feature must stay inactive. This implies that it is only useful when the feature is already activated. And a second problem arises: what happens if two implications say the opposite of each other ? The feature must be activated or deactivated ? The logic says it is an error to do that as the mapping itself becomes incoherent. This mean that uses a NOT in a consequent will always be an error of design. If a developer needs to use one of these behaviors, he will always be able to translate it in the antecedent thanks logic. As the possible changes in the consequence are meaningless or design errors, the consequent expression stay unchanged, with only the AND operator.

The accepted changes can be summarized with the new logical statements of the mapping:

$$
\begin{aligned}
&mapping ::= expression \mid mapping \wedge expression \\
&expression ::= condition \Rightarrow consequence \\
&condition ::= context \mid (condition \wedge condition) \mid (condition \vee condition) \mid \neg condition \\
&consequence ::= feature \mid consequence \wedge feature
\end{aligned}
$$

This new version allows more flexibility in the mapping. But logical expression can become difficult to handle when they are large. This point will be very important to keep in mind for the implementation. Without that, the changes could make the mapping too complex to be used easily by a developer and so become useless.

Context and Feature

Two other improvements that affect the mapping are also made. They take place in both the context and feature tree.

Historically[2], the mandatory relations acted in a passive way. All the nodes in this constraint must be activated/deactivated separately. If it is not done, the constraint is not respected and raises an error. For example, let's take a feature that represents a car which has three mandatory children, the wheels, the engine

[2]This was changed in a recent version of FBCOP thanks to the validation made in this master's thesis.

and the steering feature. To activate the car feature, the developer must submit the activation of the three children. If only the wheel and the engine are activated, the state of the model will not be valid and an error will be risen. The new approach is to consider that a mandatory entity (a context or feature) is a part of his parent. In consequence, the mandatory entity must be activated and deactivated with it, as it was a single node. So, a mandatory entity will no more have a state by itself but is fully dependent on its parent. If it receives a request (activation or deactivation), it will transmit it to its parent which will handle the request. The advantage is for both the mapping and the context changes. To activate/deactivate a node and all its mandatory children, only one entity must be activated/deactivated and no more several ones. In the previous example, to activate the car feature, the developer can now just activate the car itself (or any of its mandatory children) and all its mandatory children will be activated with it by default. This makes the mapping shorter and reduce the number of errors that the developer can make.

The second change is to make the abstract entities implicit. An abstract entity is a context or a feature which only helps to structure the other ones. For example, in the case of a weather context with different weathers as children, the weather context is abstract and its children are not. If making the abstract entities implicit does not make a big difference in the implementation, the consequences are important. An abstract entity should not be used in the next step. So an abstract context cannot be part of the mapping or an abstract feature cannot have an adaptation. Moreover a concrete entity is always a leaf of the tree in the original FBCOP implementation. This makes difficult to give an adaptation to a parent feature and impossible to use the parent context in the mapping even if it is shorter (the parent is always active if one of his children is too). Making implicit the abstract entities, allows making no differences between abstract and concrete entities. This makes all entities able to have children or to be used in the next step if needed. In the original implementation, if someone wants to develop a filter feature that can use different filters, he has to do it like on the left of the figure 3.1. The filter's behavior, needed to run a specific filter, was isolated because the parent of the specific filter cannot have an adaptation. With this proposal, the filter's behavior can be directly given to this parent as the figure 3.1 shows it on the right. This makes the feature model shorter and clearer. Moreover, as it is the parent that owns the behavior needed to run a filter, the mapping does not have to activate it.

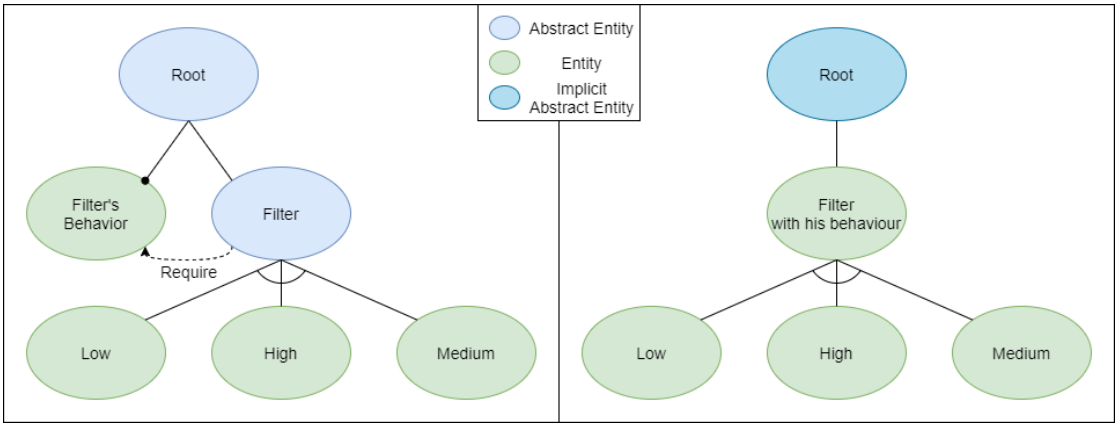


Figure 3.1: Left the architecture needed to have a parent's behavior with abstract features. Right the same case with implicit abstract.

3.5 Conclusion

This master's thesis defines a new mapping in order to answer the problem of the current one. The current mapping has a lack of expressiveness and becomes too complex when the application becomes large. Inspired by the reviews made on other mappings, a new definition for the mapping is given. The possibility to use the full propositional logic, thanks to the AND, OR and NOT operators, is given in the antecedent of the implications. But the consequent stays the same as in the current mapping. Moreover two other element that affects the mapping will be changed. The behavior of the mandatory relations, in the context and feature models, will become active and the distinction between abstract and concrete contexts or features will be erased.

Chapter 4

Implementation

This chapter will describe the implementation of the mapping. First, the new mapping declaration will be presented. Then an overview of how the current mapping works is given as a frame of reference. Next, the intuition behind the new mapping is presented followed by its architecture. Then the different parts of this architecture are detailed. In the last section, the operations done during the bootstrap are reviewed.

4.1 Mapping declaration

The mapping declaration is made by the developer with a function that returns an array of *Implications*. Each *Implication* is created with an antecedent given in the form of an *Expression* and a consequent given as a list of feature names. An *Expression* can be an operator applied to sub-expressions (*Or*, *And* or *Not*) or a context name. The *Or* and *And* operators are created with a list of *Expressions* and the *Not* with a single one.

The new version of Listing 3.1 is provided in Listing 4.1.

Listing 4.1: New mapping of car breaks: The car can break hard on a dry road but need to break softly on a wet or snowy road.

```
module MappingDeclaration
  def self.create()
    hard_break = Implication.new('Clear Weather', ['Hard Break'])

    antecedent_soft_break = Or.new('Rainy Weather', 'Snowy Weather')
    soft_break = Implication.new(antecedent_soft_break, ['Soft Break'])

    return [hard_break, soft_break]
  end
end
```

The declaration of the new mapping gives a simple and easy-to-use interface to the developer. As large logical expressions are difficult to handle, the possibility to split an expression in several sub-expressions is given to the developer. In Listing 4.1, it is the case in the second implication. Moreover, these sub-expressions often have a meaning by themselves and can thus be given a name that describes them well.

In consequence, any mapping declaration should be several, easy to understand, expressions which can be used in several other expressions or implications each.

On this small example, the advantages of the new declaration are not evident. But on huge and complex mappings, this new declaration should stay manageable by a human.

4.2 New mapping

The new mapping could follow the same flow as the current one:

The current mapping receives as input the list of context changes from the context model. It outputs the list of feature changes that will be applied on the feature model. Each context change passes, one by one, through the mapping. During this pass, all implications are evaluated in order. If an implication becomes active or inactive, the feature changes are added to a list. One list keeps all the feature activations and one all the deactivations. When all context changes are processed, the deactivation list is reverse and then, the activation list is appended to it. This list is then sent to the feature model.

Checking all implications in the mapping becomes inefficient when it becomes large. Moreover, as the conditions are more complex, they are also slower to evaluate. In consequence, the more the model becomes large, the more time is needed to apply a change. To avoid this performance degradation, the complexity of the mapping evaluation must be independent of the number of relations. A mapping with this property has already been proposed by using a Petri-net formalism [2] but this master's thesis will use another approach.

The new mapping takes its inspiration on electronic gates. Each gate knows the gate on which it depends and those which depend on it. It receives the state changes of the first ones and transmits its own changes to the second ones. Thanks that, the time complexity of the mapping will be independent of its size. In fact, only the gates that could change of state will be evaluated which is the best possible complexity for the mapping.

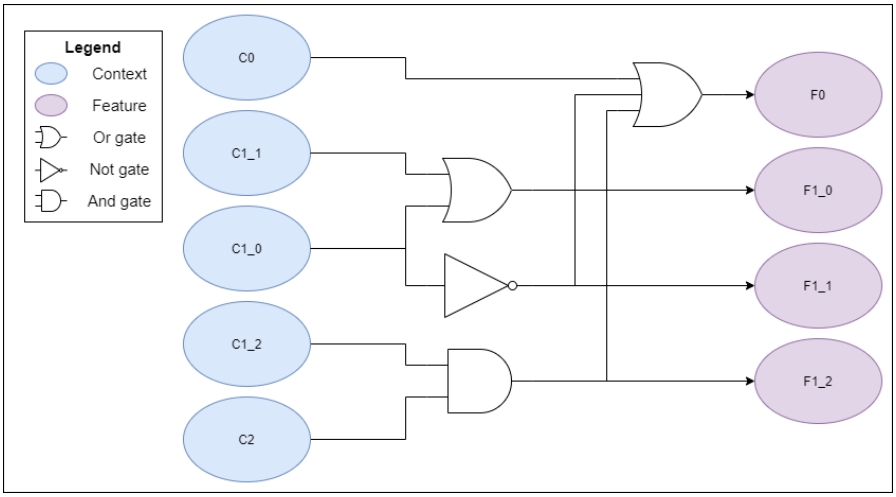


Figure 4.1: Example of mapping with gates

The figure 4.1 show the gates representation of the following mapping statement:

$$\neg C1\_0 \Rightarrow F1\_1$$
$$C1\_0 \vee C1\_1 \Rightarrow F1\_0$$
$$C1\_2 \wedge C2 \Rightarrow F1\_2$$
$$(C1\_2 \wedge C2) \vee \neg C1\_0 \vee C0 \Rightarrow F0$$

With the gate architecture, if several expressions use the same sub-expression, this sub-expression can be evaluated only once and be used by several expressions. It is the case with the gate And and the gate Not in the figure 4.1.

But this figure also shows two problems. The first one is that the Contexts and Features are directly linked to the mapping. The second one is that the order of the implications of the mapping statement is lost once represented with gates.

In the current implementation, the contexts, the mapping and the features are handled in different phases. As each phase only needs to receive changes of the previous one, it works well. But as the new mapping requires each context to know which gates use it, it becomes difficult to decouple the models. To address the problem two major solutions exist. The first is to simply create a hash map with the contexts as key and the gates as value. But this has a major drawback. The current implementation of the contexts and features are not optimized. So to change one context or one feature, the entire tree is checked. In such circumstance, having an optimized mapping for its time complexity is useless. That leads to the second solution, re-implement both trees and integrate them in a model that groups the contexts, the mapping gates and the features. Moreover it also simplifies the rollback of the model. If an error arises during a change, the complete model is rolled back. A problem of the actual implementation is when the error rose in

the features tree. As the two trees were totally independent, the changes applied on contexts stayed and only the features were rolled back. By grouping the three parts in one, it allows to rollback the contexts, features and mapping in one. This makes impossible the desynchronization of the different models.

Give the right order to the mapping is a more challenging task. A simple solution as notify the gates directly from the context or the previous gate can not work. As a same gate can be used by several implications, its change of state can not be delivered to the next gates one just after the other. In fact, to deliver a change of state to an implication, all the possible messages for another implication before this one must be delivered. In consequences, the implication itself must be represented with a gate and the messages to notify a gate must be prioritized to respect the order of the implications.

A solution is to give a priority order to all the gates according to the order of the declarations. The gates involved in an implication before another one must have a higher priority. To rank the gates used in a single implication, a sub-expression must be prioritized on the expression which uses it. If several sub-expressions are used in a same expression, the priority will be decreasing with the order of the declaration. Consequently, when a context or a gate want to transmit a message to another gate, it sends it to a priority queue where the priority of a message is the priority of its destination gate. If several messages are sent to a same gate, the first send is the first delivered. To compute the evaluation of the mapping, all the messages of the priority queue must just be treated in order.

The priority order of the gates of the example of the figure 4.1, and inferred from its logic statement order, is shown in the figure 4.2. In this figure, a repeat gate is added. Its purpose is to be able to reuse a previous gate (the Not in this case) and to respect the priority order that each gate must have. In the last implication, the And must be before the Not. They both already exist but in the opposite order. To reuse them, a simple gate that just repeat the messages of the miss-placed gate is used to simulate the needed order. The use of a repeat gate is totally transparent for the developer.

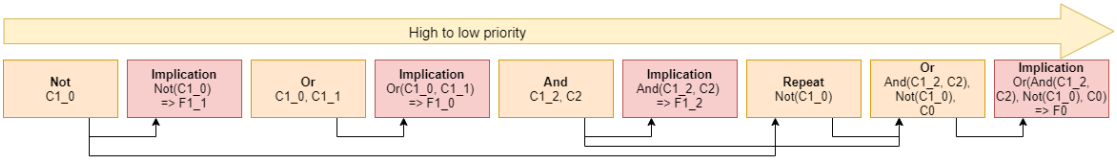


Figure 4.2: Example of gates priority

4.3 Architecture

The architecture of the global model (which groups the contexts, mapping and features) is split in five plus one parts as figure 4.3 shows. The first five parts are actually modeling a single element: the contexts, the contexts constraints, the mapping , the features and the features constraints. The last one is the controller.

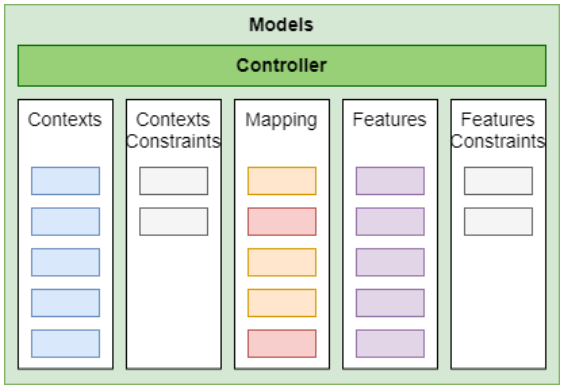


Figure 4.3: Models Architecture

Each model is composed of a list of Nodes. Any Node of a model can be reached by knowing its index or its name. A Node is any object that implements the interface needed by the controller. This interface allows the controller to handle the rollback or the commit of a new state of a Node. But also to initialize the Nodes, check the validity of their states or send messages to them. The Nodes can be split in four categories: the contexts, the features, the gates and the constraints. Each category has its own model except the constraints which has two (one for the context and one for the features).

The controller maintains a message queue for each model (a FIFO queue for all the models except the mapping which uses a priority queue). When a model is processed, all the messages for this model are sent to it in order. The controller also maintains three other lists. One for the Nodes whose state has changed, one for the Nodes which need to be checked before committing the model changes and the last one which keep the actions performed on the features. The first two lists are used to validate and commit or rollback the model changes. This last list is used to perform the activation or deactivation on the adaptations when the new state is committed. This list will be refereed later as the actions-features list.

The complete interface of these elements is shown in the figure 4.4.

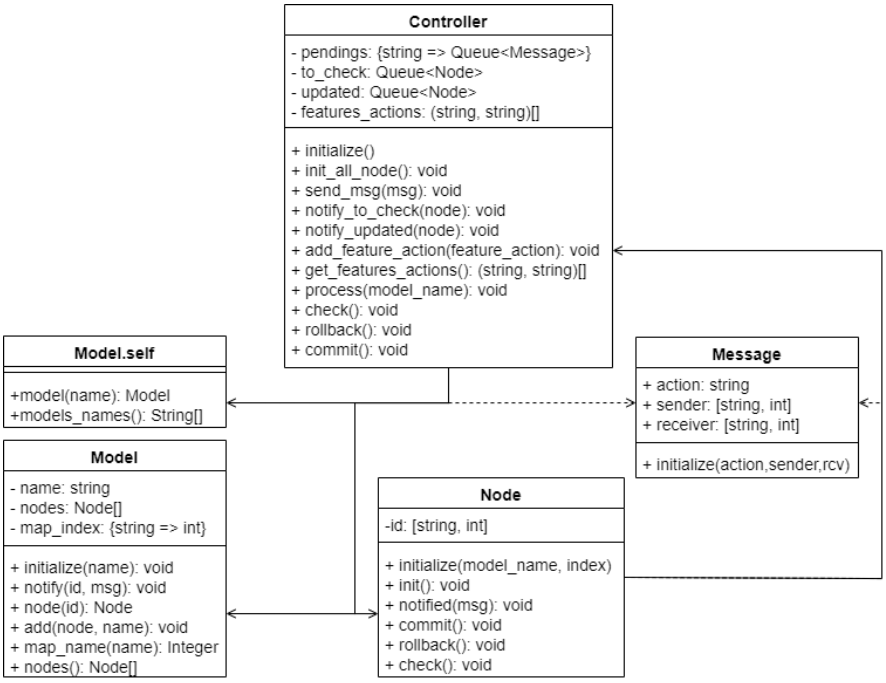


Figure 4.4: Interfaces of the model

The way to apply a context change is to send it to the controller. Then the controller must process the messages of the contexts, contexts constraints, mapping , features and features constraints in order. When it is done, the state of the model is checked and committed if valid or rolled back in the other case. If the new state is committed, the resulting actions-features list is sent to the next phase, the Feature Execution.

4.4 Model

This section will present the details of the five models involve in the architecture.

4.4.1 Contexts and Features Models

The context and feature diagrams are declared by the programmer as a tree. But the models are stored in an array. Each tree must so be flattened and each entity (a context or a feature) will be inserted in the order of a Depth-first search. So when an entity is inserted in the model, this entity will know its parent index but only the name of its children as they will be added later. The names will be replaced by the ids of the children when the controller initiates all the Nodes.

The state of an entity is composed of two counters. The first one will count the number of activations done by the exterior (a developer for the contexts or a gate of the mapping for the features). The second one will count the number of activations done by the interior (a child entity). The use of two counters avoids to deactivate an entity (counter equals to 0) if a child is still active. With a single counter, this is possible, for example if the developer active a child and then deactivate its parent. To handle the rollback of the Node, these two counters are duplicated in a pending and a committed one. All the changes to the counters are made on the pending counters. If the model is committed the values are copied to the committed counters but if the model is rolled back, the pending values are reset to the committed ones. An entity is active if one of its two counters is positive and inactive if both are equal to zero. Negative numbers are tolerated in the pending counter, but can never be committed. So if the value is at a moment negative, the entity will subscribe to the check list of the controller to be checked before committing the model and and the entity will raise an error if the value is still negative at this time.

The context and feature model are both composed of two kinds of Nodes. Every entity (a context or a feature) can be optional or mandatory. An optional entity is not affected when its parent is activated whereas the mandatory one share its parent's state.

The activation of an optional entity is performed as follows. Before doing anything else, the entity notifies the controller that it is updated. In the case of an activation done by a child, the entity propagates the activation to its parent before updating his state. If the entity becomes active, it then notifies its subscribers (every Node which need to know its state) and then activates its mandatory children. In the case of an external activation, the entity updates its state first and, if it becomes active, activates its parent, notifies its subscribers and then activates its mandatory children. The code of these two activations is given in the Listing 4.2 The deactivation is performed in the exact same way.

Listing 4.2: Code of the activation of an entity

```
def activate(sender, intern_call=false)
  notify_updated()
  if intern_call then
    intern_act()
  else
    extern_act()
  end
end

def intern_act()
  parent_act()
```

```
    if !activated? then
      notify_subscriber(:activate)
      mandatory_children_act()
    end
    @pending_child_count += 1
  end

  def extern_act()
    if @pending_self_count == 0 then
      parent_act()
      if !activated? then
        notify_subscriber(:activate)
        mandatory_children_act()
      end
    end
    @pending_self_count += 1
  end
```

In the case of a mandatory entity, all activations or deactivations performed by the exterior or the interior are transmitted to its parent as if the call was performed directly to it. So an external activation is sent as external activation to its parent and not as an internal one. This is the consequence that a mandatory entity does not have its own state but shares its parent state. When the parent becomes active/inactive, it calls a specific function of its mandatory children to activate/deactivate them. This specific function first notifies the controller that its state has changed, then notifies its subscribers and then activates its mandatory children.

The only distinction between the contexts and the features, is that the feature calls the controller when it notifies its subscribers to add its change of states on the list of actions-features.

4.4.2 Constraints Models

The passive constraints of the context and feature models do not add any behavior to the context or feature entities but raise an error when they are not respected. In consequence, they are not parts of the context or feature models but have their own models. On opposite, the active constraints, the *Optional* and *Mandatory*, have their behavior implemented in the entities and so are not part of the constraints models.

Four passive constraints exist on these models. The first two ones have been already presented (seen in 2.2.1). This is the *Alternative* and the *Or* constraints. They model the constraint between an entity (the source) and its children (the

targets). Two other exist, the *Require* and the *Exclude*. They can constrain one entity (the source) with any other entities (the targets). The *Exclude* constraint forbids having the source active at the same time as one or several of its targets. The *Require* is its exact opposite, i.e. the source need, to be activated and stays active, all its targets active. This last constraint is the only one with an order, as the source require that the targets are active at the moment of its activation and stay active until its deactivation. This implies that this constraint must always be respected, even during the model changes. The other constraints can have an invalid state during the changes as they do not involve the order to be valid at the end of the model changes.

When a constraint is created, it is added to the constraint model affiliated with the entities it constrains. Then, it subscribes to the changes of its source and targets.

The state of all the constraints has two values. One to know if its source is active or not. The second that keeps the number of active targets. Of course as for the entity, two versions of these values exist to be able to make a rollback.

When a constraint is notified of a change, it first subscribes to the changed list of the controller. Then it updates its state according to the change. If the state is invalid, it subscribes to the check list of the controller to be checked later or, in the case of the *Require*, raise an error.

The valid state of each constraint is described in the Listing 4.3. The pending_active represents if the source is active or not, and the pending_count the number of active targets. The valid function is called to verify the state of a constraint, at the end of its update state or when the controller checks the constraint later.

Listing 4.3: Valid state of the constraints

```
#Alternative
def valid?()
  return !@pending_active || @pending_count == 1
end

#Or
def valid?()
  return !@pending_active || @pending_count > 0
end

#Exclude
def valid?()
  return !@pending_active || @pending_count == 0
end
```

```
#Require
def valid?()
  return !@pending_active || @pending_count == @targets.length
end
```

4.4.3 Mapping Model

The mapping model is composed of gates. The index of a gate is given according to its priority level. The gates with the higher priority are inserted first in the model and so will have a low index, whereas the gates with a low priority will be added last and so will receive a higher index. This order allows a gate to subscribe to all needed sub-expressions when it is added to the model as all of them are already part of the model. The index 0 will always be the implication between the root of the context and the root of the feature model.

There exist four types of gates.

The implication is the simplest one. This gate simply retransmits the received messages to its subscriber. In consequence, this gate has no internal state. This gate is also used as a repeat gate to reorder the sub-expression of a gate. When this gate is created as an implication, the features of its consequent are added to its subscribers in the order of the declaration. The first will be activated first, and the last one at the end.

The second gate is the Not. As the implication, it has no internal state. This gate simply retransmits the messages it receives but with the opposite action. So an *:activate* become a *:deactivate* and inversely. When the model is initiated, this gate will emit a message of activation to all its subscribers. This comes from the fact that every Node is considered to be inactive at the start.

The last two gates, the Or and the And, share the same behavior. They both have a state that counts the number of active sub-expressions. In consequence, its state is duplicated to allow the rollback and at each change they must subscribe to the changed list of the controller. The Or gate will be active when its counter is positive. And, the And gate will be active only when its counter is equals to its number of sub-expressions.

4.5 Models bootstrap

The bootstrap of the models creates all the models and gives their initial state.

Firstly, the declarations made by the programmer will be translated to create the five models. In order, it creates the context and context constraints model, then the feature and feature constraints before ending with the mapping. When it is done, the controller contacts each Node to run its initialization function.

In the context declaration, the developer can specify if some gates must be activated during the initialization. By default only the root is activated with all its mandatory children. These changes are thus submitted to the model. The resulting actions-features list is then transited to the feature execution to deploy the needed adaptation.

4.6 Conclusion

A new mapping declaration is proposed which is more flexible and expressive than the current one.

The new implementation merges the context model, the mapping and the feature model in a single model. This not only optimize the three parts but also avoid the desynchronization of them and make the rollback mechanism simpler. This model is divided in 6 parts, the controller and five sub model: the contexts, the context constraints, the mapping, the features and the feature constraints. The new mapping is implemented with logic gates that have a priority order inferred from the declaration.

Chapter 5

Transaction

This chapter will present the transaction based system. The transaction allows the developer to change the context and so the behavior of the system. The chapter starts with the description of a transaction. Then, different strategies will be presented. Afterward the possible properties for a transaction will be given. Then the properties of the different strategies will be presented and proven. Finally, a discussion of the strategies is made.

5.1 Description

The transaction is the mechanism given to the developer to submit context changes during the execution of a program. The transaction starts with the submission of a list of instructions which represent each a change of contexts, continues with the update of the state of the model and ends when all the associate adaptations are deployed or removed. After a transaction, the program continues its execution with its new behavior.

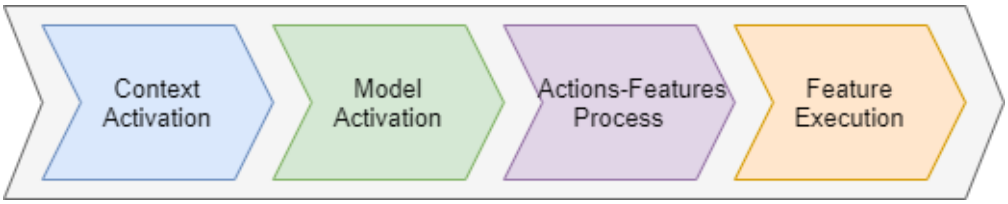


Figure 5.1: The steps of a transaction.

As the figure 5.1 shows it, a transaction involves the following steps in order:

1 - Context Activation

This step is directly called by the programmer by submitting to it a list of instructions.

The role of this step is to offer an interface to the developer and to protect the other steps. This protection prevents the constraints of the models from being violated by the developer or from executing several transactions in parallel.

If an error rise during a transaction, this step will intercept it and rollback the model. But this will only cover the model and not the feature execution as

it does not implement a rollback mechanism. So if an error rise in this last part (which normally never happens), the model can be de-synchronized from the feature execution. If this happens, the program must be stopped as it will create errors in the next transactions.

A second protection avoids that a programmer run several transactions in parallel as the models are not built for it. To avoid this, a mutex allows only one transaction to run at a time. If several are submitted in parallel, the first one will run until completion, then the second,...

2 - Model Activation

This step submits the instructions to the model and propagates the changes in it.

Two possibilities to submit the instruction to the model exist. This can be done sequentially or simultaneously. The sequential submission is the one used in the original implementation. It submits the first instruction and process the entire model, then the second, and so on until the last one. When all the instructions are submitted and processed, the model is checked and committed. In the case of a simultaneous submission, all the instructions are submitted to the model in once and then the model is processed. The following parts stay the same as in the first case.

The distinction between these two strategies is in the order of activation of the features. The final state of the models in the two strategies is the exact same one. But as the feature execution result depends on the order of the activation, this difference is important. In the case of the sequential submission, the order of the instructions is applied first and then the order of the mapping. This is equivalent to say that the implication of the mapping are evaluated with the changes of the first instruction, then with the second,... In the other case, it is the order of the mapping that is applied first and then the order of the instructions. This time, the equivalent is to say that the implications are evaluated only once with the changes of all the instructions.

Let's take a simple example to illustrate this:

$$C1 \Rightarrow F1$$
$$C1 \wedge C2 \Rightarrow F2$$
$$C2 \Rightarrow F3$$

Let's consider that the instructions submitted by the developer in a transaction are: "*activate C2*" then "*activate C1*".

In the case of the sequential submission, the context C2 is activated and propagated through the mapping. This activates the feature F3. Then the context C1 is activated and propagated. This activates the feature F1 and then the feature F2. The resulting order of the feature activations is F3, F1 and then F2.

In the case of the simultaneous submission, the context C2 and the context C1 are activated before propagating the changes to the mapping. The mapping evaluates the implications and activate F1, then F2 and at the end F3 according to the mapping order.

The results of these strategies are both valid. At this point of the rapport, there is no reason to privilege one on the other. The two strategies just have different behaviors.

3 - Feature-Actions Process

This step modifies the list of actions-features that go out of the model. This is done to match the properties wanted for the transaction. This will be seen deeper later.

4 - Feature Execution

The feature execution executes the feature's changes, that it receives in the actions-features list, on the adaptations that depend on these features. The activations/de-activations are performed in the order of the actions-features.

5.2 Strategies

This master's proposes four strategies. These strategies differs on two points. The first one is the strategy uses for the model (sequential or simultaneous). The second one is the function used in the Actions-Features process.

5.2.1 All instructions in one

The all-instructions-in-one strategy (all-in for short) is the simplest and the default strategy.

As its name let it know, the all-in uses the simultaneous strategy for the model. It means submit all the context changes in once to the model. And then propagates all of them in a single pass through the model.

The feature-actions process works as follows:

Firstly, all the features with several occurrences in the list are reduced to only one or zero occurrence. If the number of activations is equal to the number of deactivations for a same feature, all the occurrences of this feature are discarded. It is equivalent to say as the initial state is the same that the ending one, nothing must be done. If the number of activations is bigger than the number of deactivations, only the last occurrence of this feature is kept and all other are discarded. This

last occurrence is always an activation as a feature never emit two same actions one just after the other. If it is the number of deactivations that is the bigger, only the first occurrence is kept which is always a deactivation for the same reason.

Secondly, the activations are separated from the deactivations. The deactivation list is reverted. And the two lists are merged with the deactivations first, followed by the activations. This new list is then transmitted to the feature execution.

5.2.2 Every instruction as a step

The every-instruction-as-a-step strategy (by-step for short) is the closest strategy from the transaction used in the original implementation.

This transaction allows doing some reordering of adaptation. Let's take the following example of mapping:

$$C1 \Rightarrow F1$$
$$C2 \Rightarrow F2$$
$$C1 \vee C2 \Rightarrow F3$$

In this example F3 require F1 or F2 to work. Let's assume that C1 is already active and that the following transaction is done: deactivate C1, activate C2. With the old strategy, F1 is deactivated and F2 activated but above F3 and not below it, in the order of the proceed, which will make an error at execution. This new strategy applies the context changes one by one in the model. So in a first time C1 is deactivated and so the features F1 and F3 with it. Then C2 is activated and so F2 and F3 with it. The deactivation and reactivation of F3 allows that F2 is below F3 for the proceed. This deactivation and reactivation are called reordering as it can modify the order of the adaptations.

This strategy uses the sequential submission of the instruction to the mapping. Each instruction passes, in order, through the model alone and generate a list of features-actions. Each of these lists is treated as the one generated in the all-in strategy. Each of these list is appended to the previous one. When all instructions are passed, the resulting list is transmitted to the feature execution. In consequence, the adaptations goes through all the states generated by the states of the features between the instructions.

5.2.3 All instructions in one with reordering

The all-instructions-in-one-with-reordering strategy is a modified version of the all-in strategy which allows the reordering of alterations.

This strategy allows reordering inside an instruction where the by-step strategy only allows the reordering between the instructions. This is interesting in a case like the following one:

$$C1 \wedge C2 \Rightarrow F1$$
$$\neg C1 \wedge C3 \Rightarrow F2$$
$$(\neg C1 \wedge C3) \vee (C1 \wedge C2) \Rightarrow F3$$

In this example F3 require F1 or F2 to work. Let's assume that C2 and C3 are already active and that the following transaction is done: "*activate C1*". With the two previous strategies, F2 is deactivated and then F1 activated. So F1 is called above F3 in the proceed order and it will make an error at execution. By taking count of the order of the expression evaluation (sub-expressions are evaluated left to right), the last implication deactivates F3 and then reactivates it in this situation. This allows deactivating F3 activating F1 and then reactivating F3. [1]

The difference is in the features-actions process function. If, in the all-in strategy, the features with a number of activations equals to the number of deactivations are always discarded, it is no more the case in this strategy. In this specific case, if the first occurrence of the feature is a deactivation and the last one an activation, these two elements are kept but the possible occurrences of the same feature between these two elements are discarded. In all other cases, the same rules that for the all-in strategy will apply.

5.2.4 Every instruction as a step with reordering

The every-instruction-as-a-step-with-reordering strategy is the same as the by-step strategy but use the features-actions process function of the all-in-with-reorder and no more the one of the all-in strategy. The difference between this strategy and the by-step strategy is that the reordering is not only done between the instructions, as in the by-step, but also inside each of them.

5.3 Properties

A transaction should follow some properties. Some of them are needed to ensure the success of it. Other ones ensures that the behaviors given to the developer is respected for each step involve in the transaction.

The different properties are:

Atomic

This property assures that a transaction is either performed completely or has no impact. Thanks to this property, the ending state is guaranteed to be valid.

[1] This reorder not appears when C1 is deactivated. To handle the reordering in the two cases, a more complex mapping is needed but it is possible to do it.

Unfortunately, this property is not respected in the Feature Execution when an error occurs there. But the respect of the next property ensures that it never happens.

Error free

A transaction should not add any errors by itself. This means that if the final state of the model is valid, the transaction must be performed without errors.

Assuming there is no bug in the implementation of the model and feature execution, the errors can come from two origins. The first and unavoidable one is the code given by the developer. This kind of error should be reported to the developer without stopping the feature execution. The second source is trying to activate an active adaptation or deactivate an inactive one. These errors come from the feature-Actions Process and never happens if this step is well design.

Order respect

The order of the instructions and of the mapping must be respected (according to the strategy used to go through the model). This property ensure that the final state of the alterations is the same as the one predicted according to the declaration and so predictable by the developer.

If the model implements the activation order by design, the deactivation order is not. To get the correct order for the deactivation, the output of one pass through the model must be inverted. In the sequential strategy, it means that the deactivations, in the output of the model for a single instruction, are inverted. But the order between the outputs of the model, one for each instruction, must be kept.

Reversible

If a list of instructions is applied in a transaction and then its exact opposite in the next one, the actions-features lists submitted to the feature execution are also the exact opposite of each other. This property ensure that the state of the adaptations can be reverse with the minimum of change between before and after these two transactions. It should be noticed that it not means that the state of the adaptations will be exactly the same one. This property also makes the transactions easier to understand and more predictable for a developer.

Both instructions and actions-features list are a list of activations and deactivations in a given order. The opposite list is the list in the reverse order where activations and deactivations are inverted. For example:

$$[[deactivate, F1], [deactivate, F2], [activate, F3], [activate, F4]]$$

and

$$[[deactivate, F4], [deactivate, F3], [activate, F2], [activate, F1]]$$

Constraints respect

This property ensures that all the constraints are always respected by the adaptations, even during the transaction. This property is important when the changes also applied to other threads that are running during the transaction[2]. The order respect property assures the respect of the constraints of type: "A require B to work". But another type of constraint exists: "A can not work with B". To assure these constraints, the deactivation of the features involves in one of these constraints must be performed before the activation of another feature involves in the same constraint.

A stronger version of this property and easier to apply is all deactivations must be performed before the activations.

Stable

The oscillations of an adaptation (activation and then deactivation or inverse with eventually with something else between the two) should be discarded in a transaction.

This property is not essential but simplify the behaviors of the transaction and make it more easily predictable and so more accessible to a non expert developer. More important, its respect make easy the applications of the other properties.

The disadvantage of this property is that a perfect stability not allows the reordering of adaptations. In consequence most of the strategies use a weakened version of this property.

5.4 Strategies' properties

The four strategies respect some of the properties. The figure 5.2 shows which property is respected by which strategy.

[2]The initial implementation of FBCOP is not compliant with multithreading. But the new implementation of the Feature Execution make it possible. These changes are presented in the chapter 7.

	Atomic	Error Free	Order Respect	Reversible	Constraint Respect	Stable
All_In	✓	✓	✓	✓	✓	✓
By_Step	✓	✓	✓	✓		
All_in with reordering	✓	✓	✓		✓	
By_step with reordering	✓	✓	✓			

Figure 5.2: All the transactions with their properties

Let's proves the respect or disrespect of the properties by the transactions.

5.4.1 All instructions in one

- **Stable** : As the feature-actions process keeps only one occurrence of each feature in the feature-action list, this strategy is perfectly stable.
- **Order Respect** : As the activations are made in the same order as the output of the model and the deactivation in its reverse order, the order is respected.
- **Constraints respect** : As this strategy makes all the deactivations before the activations, the stronger version of this property is followed.
- **Error free** : This strategy keeps at most one feature-action per feature, and keep the action that occurs the most time on the output of the model. In consequence, this transaction will never activate an already activated adaptation, or deactivate an already deactivated one. These errors are so impossible.
- **Atomic** : As the strategy follows the error free property, the atomicity of the transaction is ensured.
- **Reversible** : This strategy only keep the features-actions that represent the changes of the features states between before and after the transaction. The order of these features-actions only depends on the order of the mapping as only one pass through the model is done. This strategy is so independent of the order of the instructions given to the transaction. In the reverse transaction, the order of the mapping is not changed but the instructions are

inverted. Thus, the output of the model after removing the oscillations is the same in the reverse transaction as in the initial one with all actions inverted. Then, the deactivations are reversed and putted before the activations. In consequence, the first deactivation in the initial transaction becomes the last activation in the reverse transaction and the last activation becomes the first deactivation. As the features-actions list of a reverse transaction is the opposite of the initial one, the transaction is reversible.

To summarize, the all-in strategy respects all the properties.

5.4.2 Every instructions as a step

- **Stable** : This strategy is not stable. If someone submit two opposite instructions (the activation and the deactivation of a same context) in a transaction, this activates/deactivates some feature in the first instruction and the second does the opposite. This comes from that each instruction can be seen as an all-in strategy and this strategy is reversible. In consequence, all the features in the features-actions list oscillate in this specific transaction.
- **Order Respect** : As the order of the output of each pass in the model is respected as it is equivalent to the all-in strategy and that lists are grouped in the order of the instructions, the order is respected (according to its definition for the sequential submission).
- **Constraints respect** : This property is not respected by this transaction. This comes from the fact that an activation can be performed before a deactivation even if the two features are constraints with an alternative or an exclude.
- **Error free** : As this transaction is equivalent, for the feature execution, to submit each instruction in a transaction with the all-in strategy, this transaction will also be without the errors of activation/deactivation of the adaptations. This property is so respected.
- **Atomic** : As the strategy follows the error free property, the atomicity of the transaction is ensured.
- **Reversible** : This transaction can be seen like several transaction applying the all-in strategy with one instruction. Thus, the reverse transaction is equivalent to reverse the last instruction, then the previous one,... As the all-in strategy is reversible, this strategy is also reversible.

To summarize, this property follows all the properties except the Constraint Respect and the Stability.

5.4.3 All instructions in one with reordering

Only the properties where the proof changes from the all-in strategy are described. The other ones stay valid. These unchanged properties are the Atomic, the Order Respect and the Constraint Respect.

- **Stable** : This strategy is not stable. But only one oscillation per feature is allowed if it can help for the reordering of the alterations.
- **Error free** : As in the couple of actions for the same feature, respect the order of the output of the model, a deactivation first then a re-activation, these features-actions cannot create an error. So this property stay valid in this strategy as the other actions-features, kept by the Action Feature Process, are the same as in the all-in.
- **Reversible** : This property is not respected in this strategy. If the final state of the model can always be reverted to its original state, the path it takes is not guaranteed to be the exact same one. In fact, only the oscillations of the model done in one transaction can change in the reverse transaction. But as this strategy keeps some of them, the reversibility is lost.

To summarize, this property follows all the properties except the Reversibility and the Stability.

5.4.4 Every instruction as a step with reordering

The properties of this strategy are the same as the ones of the by-step except it loses the reversibility for the same reason as the all-in-with-reorder. In consequence, this strategy only follows the property needed to guarantee its success, the atomic, the Error Free and the Order Respect.

5.5 Discussion

Four strategies were presented. But in which cases the developer need to use a specific strategy? And are them all useful?

- **All-In** :

  This strategy is easy to understand and, except when reordering is needed, will always do the job. Moreover, thanks the order and constraint respect, this strategy can be used with parallel execution. Indeed as the constraint are respected even during the transaction, the other threads will always use a valid state of the adaptations.

- **By-Step** :

  The by-step strategy is also an easy-to-understand strategy. This is the consequence of that this strategy only passes by the state between the instructions and do not care of the order of sub-expressions in an implication. So, the reorder is simple to design and predict. In most of the case, this transaction is sufficient for the wanted reordering.

  As the constraint respect property is not valid, this instruction could create errors with parallel executions.

- **All-In-with-reorder** :

  The all-in-with-reorder allows every kind of reordering. This means that even with a single instruction and a well design mapping, the reordering is possible. But this strategy needs the developer to very well understand how the mapping work. Without that it could create some surprise at execution. This strategy is only advised for expert developers.

  The second advantage is that this strategy is compliant with parallel execution. It allows making reordering safely in this case.

- **By-Step-with-reorder** :

  The by-step-with-reorder strategy mix all the disadvantage of the by-step and all-in-with-reorder. It is the strategy with the more freedom, and so, the one that respects the less number of properties. In brief, there is no known use case for this strategy.

The first three strategies have different behaviors and are addressed to different kinds of developers. In consequence, the three must be kept. The last one has no real application. Its use is not recommended as it does not bring any new behavior but has more disadvantages than the other one.

5.6 Developer Interface

Three functions are provided to the developer to perform the transactions. All are accessible through the *ContextsActivation* class. These functions are equivalent to the ones of the original implementation but with the possibility to choose the strategy for the transaction.

Activate

Activate one or several contexts in the given order. The strategy used can be specified at the end.

```
ContextsActivation.activate('Context1', 'Context2',
    ..., :strategy)
```

Deactivate

Deactivate one or several contexts in the given order. The strategy used can be specified at the end.

```
ContextsActivation.deactivate('Context1', 'Context2',
    ..., :strategy)
```

Toggle

Activate and/or deactivate one or several contexts in the given order. The strategy used can be specified at the end.

```
ContextsActivation.toggle([:activate, 'Context1'], [:
    deactivate, 'Context2'], ..., :strategy)
```

5.7 Conclusion

Every transaction pass through four main steps: the context activation, the model activation, the action features process and the feature execution. This master's thesis introduces four different strategies with different behaviors. These strategies are evaluated by six properties (Atomicity, Error Free, Order Respect, Constraint Respect, Stability and Reversibility). Only the all-in strategy respect all properties, the by-step and the all-in-with-reordering strategies violate some of them to bring new behaviors. The last strategy, the by-step-with-reordering only respect the first three properties which are mandatory to ensure the success of the transaction. This last strategy is not recommended as it does not add something new to the other ones and only have more disadvantages. The three other strategies are interesting and must be chosen according to the application and to the understanding of the language by the developer who uses it.

Chapter 6

Validation

This section will validate the new version of the mapping model. It will first present the case study used for the validation. Then the expressiveness of the mapping will be discussed. Next, the performance of the new mapping model will be evaluated and compared to the old version.

6.1 Case study: COP Car

Nowadays, all the cars use a lot of electronics to handle everything. This case study is to implement a COP software for an automatic car. This software must handle the behaviors of the breaks, the acceleration, the gears,... But the car must also respect the speed limitation, adapt its behavior to the ground, and so on. With the contexts and features of the car, all of that is handled by the mapping of this FBCOP software.

Let's now see which contexts and features this car uses.

6.1.1 Contexts

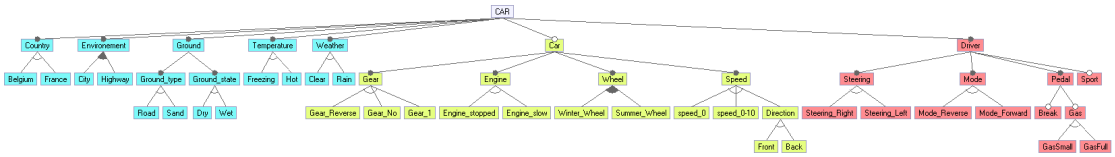


Figure 6.1: Simplified version of the Contexts of the car (complete version in appendix A)

The context model is composed of three categories. One that describes the driver inputs (in red), a second that describes the state of the car (in yellow), and the last describes what is around it (in blue). A simplified version of the contexts can be seen in the figure 6.1 and the complete version in appendix A.

The first category is about the driver inputs. It describes the state of the steering wheel and the pedals. But also if the driver selects to go forward, backward or to stop the car. The sport mode allows the driver to drive as he wants with maximum performance but with no respect for speed limitations.

The second category describes the car state. It represents its speed, the speed of its engine, its wheels, and which gear is being used.

The last category describes the global environment. It can be quite general as the country where the car is or more local such as if there are a school nearby or if the car is in a city, in the countryside or on a highway. These contexts also describe the ground where it rides, the temperature and the weather.

6.1.2 Features

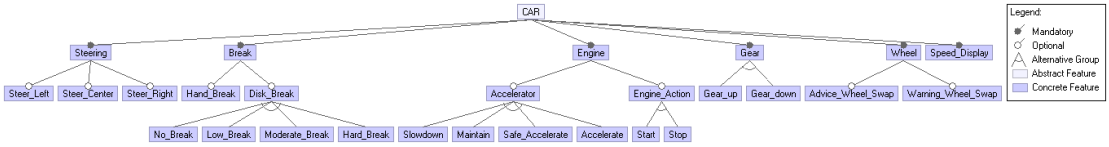


Figure 6.2: Features of the car (larger version in appendix B)

The features control the car. It means that when a feature is active, the car is applying it. For example, if the feature *hard_break* is active, the disks of the car are being used to slow down the car.

All the features are represented in the figure 6.2 or in appendix B for more readability. Let's describe most of them and their use. The *steering* feature includes three sub-features as the steering contexts to assist the driver to turn the wheels: *left*, *right* and *center*. The *break* has two main features, the *hand* and the *disk* breaks. The *hand break* will be used when the car is stopped. The *disk break* has different possible behaviors to break in every situation. The *hard break* will be used on a dry road, the *medium* when it is wet or on dry clay. The *low break* will be used in more extreme condition like snow, ice , mud or sand. The *engine* has two kinds of features that control it. The *actions* that can start or stop it and the *acceleration* that controls how it runs. The engine will slow down when the driver does not push on the acceleration pedal or when the speed limit is exceeded. The *maintain* allows keeping the same speed and is used when the driven push a little on the pedal. When the driver wants a full acceleration, one of the two *accelerations* of the *engine* will be used according to the situation. The gears will have the features to pass the *gear up* or *down*. The *wheels* will give a warning when they are not adapted to the situation. The last feature is simply to *display the speed* of the car.

The features can make a punctual action like start or stop the engine or pass a gear. They can also be contiguous as for the breaks or the acceleration of the engine.

6.2 Mapping Expressiveness

The new declaration of the mapping is more expressive than the old one thanks the possibility to use the OR and NOT operators in addition to the AND that already existed. But is it really useful on a real application? And even if it is the case, is this new declaration easily writable and readable?

To answer these questions, the experiment is to try to design a complex mapping with the new declaration. So the COP car case study will serve for that. The complete mapping declaration can be found in the appendix C.

The part of the mapping that handles the breaks is shown in the Listing 6.1

Listing 6.1: Mapping of the breaks in the COP Car case study

```
#ground
off_road = Not.new('Road')
not_freezing = Not.new('Freezing')

ice       = And.new('Wet', 'Freezing')
mud       = And.new('Clay', 'Wet', not_freezing)
dry_sand = And.new('Dry', 'Sand')
icy_road = And.new('Road', ice)
wet_road = And.new('Road', 'Wet')

very_slippy = Or.new('Snowy', dry_sand, mud, icy_road)
slippy       = Or.new(wet_road, off_road, very_slippy)
grippy       = Not.new(slippy)

#speed
stopped      = 'Speed0'
moving      = Not.new(stopped)

#unrespected_mode
not_reverse = And.new(moving, 'Front', 'ModeReverse')
not_stopped = And.new(moving, 'ModeStop')
not_forward = And.new(moving, 'Back', 'ModeForward')
unrespected_mode = Or.new(not_reverse, not_stopped, not_forward)

#breaking
hand_break      = And.new('ModeStop', stopped)
active_break    = Or.new('Break', unrespected_mode)
no_break        = Not.new(active_break)
low_break       = And.new(active_break, very_slippy)
moderate_break = And.new(active_break, slippy, Not.new(very_slippy))
```

```
hard_break      = And.new(active_break, grippy)

rule_hand_break      = Implication.new(hand_break, ['HandBreak'])
rule_no_break        = Implication.new(no_break, ['NoBreak'])
rule_low_break       = Implication.new(low_break, ['LowBreak'])
rule_moderate_break = Implication.new(moderate_break, ['ModerateBreak'])
rule_hard_break      = Implication.new(hard_break, ['HardBreak'])
```

Already on this small part of the mapping, all the operators (And, Or and Not) are used and necessary to model the complexity of the relations between the contexts and features. This mapping would be impossible to express in the original mapping declaration. This is the consequence of the impossibility to express the negation of a context. But with only the low_break implication and the introduction of a new context 'Moving' to avoid the Not, the mapping of this car with the original declaration can be found in the Listing 6.2

Listing 6.2: Mapping of the low break implication from the COP Car case study given in the original declaration

```
mapping = {
['Snowy', 'Break']
 => ['LowBreak'],
['Snowy', 'Moving', 'Front', 'ModeReverse']
 => ['LowBreak'],
['Snowy', 'Moving', 'ModeStop']
 => ['LowBreak'],
['Snowy', 'Moving', 'Back', 'ModeForward']
 => ['LowBreak'],
['Dry', 'Sand', 'Break']
 => ['LowBreak'],
['Dry', 'Sand', 'Moving', 'Front', 'ModeReverse']
 => ['LowBreak'],
['Dry', 'Sand', 'Moving', 'ModeStop']
 => ['LowBreak'],
['Dry', 'Sand', 'Moving', 'Back', 'ModeForward']
 => ['LowBreak'],
['Wet', 'Freezing', 'Road', 'Break']
 => ['LowBreak'],
['Wet', 'Freezing', 'Road', 'Moving', 'Front', 'ModeReverse']
 => ['LowBreak'],
['Wet', 'Freezing', 'Road', 'Moving', 'ModeStop']
 => ['LowBreak'],
['Wet', 'Freezing', 'Road', 'Moving', 'Back', 'ModeForward']
 => ['LowBreak']
}
```

It must be noticed that the use of the name of the contexts and features was not possible in the original mapping declaration. But to keep this example short, this freedom has been taken. This example shows that even when the expressiveness of the new mapping is not required, it is extremely challenging to express or to read a complex rule in the original declaration.

My personal feedback as COP developer is that the new declaration is easily manageable and writable even when the rules become extremely large thanks to the possibility to define intermediate expressions and to reuse them later. This creates a large number of small and easy-to-understand expressions. If a single line of the mapping declaration is taken, it stays possible to understand it without reading the previous lines. This is very useful to avoid the errors or to debug the mapping.

Of course, this case study only proves that, in this specific case and for one developer, this new declaration is useful. But this new mapping declaration seems to give a more adapted expressiveness and a better clarity to the code.

To get a more interesting result, the old and the new mapping should be proposed to make an application to a wide number of programmers. And if the majority agrees to say the new one has a better expressiveness and clarity, then it could be considered that these changes are useful.

6.3 Model Performance

The performances of the model is evaluated for two factors, the size of the model and the deepness of the mapping. For the first factor, the performance of the new implementation is compared to the original one. This is not the case for the second factor as the original implementation has always a deepness of one.

Factor: size of the model

This experiment will show the average time of one transaction with a single instruction in function of the size of the model. It will also compare the performances between the new implementation and the original one.

A model of size N has the following shape: N contexts which are the direct children of the root context, N relations that link one context with one feature and N feature which are the direct children of the root feature.

The test is run as follow: One context is chosen randomly between the N ones. In a first transaction, the context is activated. In a second one, it is deactivated. This is then repeated several times.

The values generated are the average times calculated with the measures made on 200000, 20000 or 2000 transactions receptively for the model sizes between 0 and

32, between 64 and 1024 and between 2048 and 32768 for the two implementations. A warm-up period of respectively 20000, 2000 and 200 transactions is run before.

The average times are calculated for the mapping, the contexts, the features and the complete model. The complete model is considered as being all the transaction steps except the Feature Execution in both implementations. As the new implementation makes some operations for the entire model, they are part of the model time but not of the mapping, contexts and features. But even if the time cannot be compared directly between the new and original implementation of these three elements, their time complexity can be compared graphically.

The results of the mapping, the contexts, the features and the complete model are shown in figure 6.3.

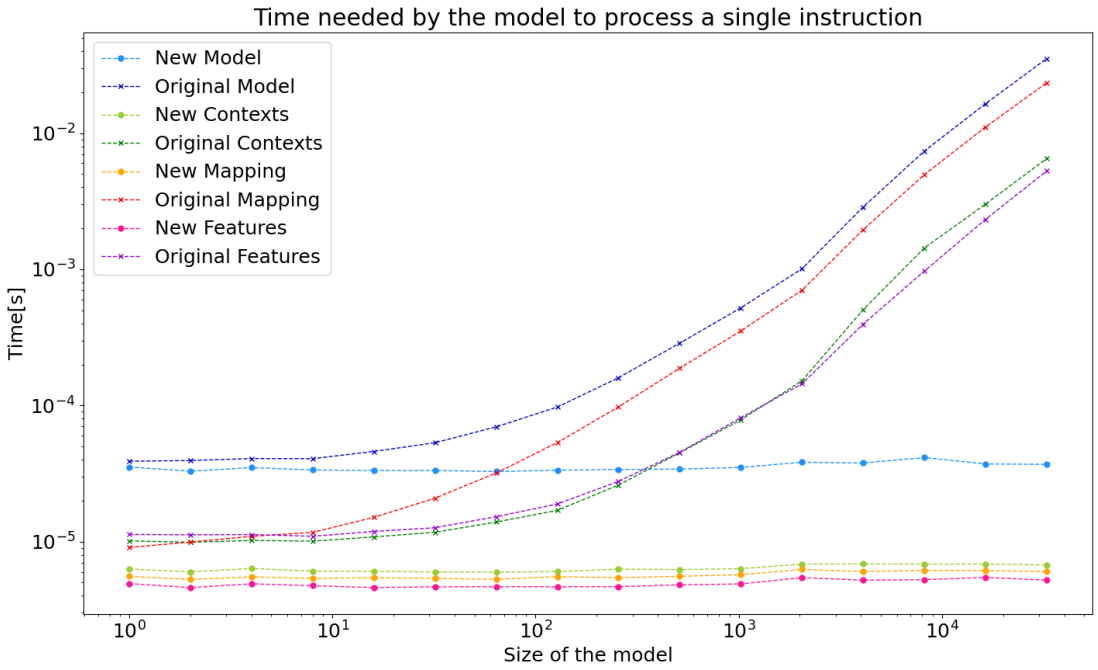


Figure 6.3: Comparison of the new and original implementations on the time needed to process an instruction

This figure shows that the new implementation is well independent of the size of the model. This can be seen in the figure 6.3 as the times needed by the new implementation are constant no matter the size of the model. The original one has the complexity of all its elements that depend on it, as more the size of the model is large, more the time needed to process an instruction is long.

On a very small model, the new and the original implementation nearly have the same performance. The performance of the original implementation is slightly

degraded until the model size of a few hundreds. After this point, its linear complexity hardly slows down the transactions.

Factor: Deepness of the mapping

The antecedent of an implication can be seen as a tree of expressions. The deepness of the mapping is the deepness of this tree.

The model is tested with two kinds of mappings. One that simulates the worst-case possible for a given deepness (symbolize by N) and one that represents a more realistic mapping which represents an average case.

The size of the model is fixed to 1024 contexts and 1024 features. The worse case is modeled as a 1024 implications with N consecutive Operators (AND, OR or NOT) for a context. So for N equals two and with the operator And, the antecedent will be And(And(context)). Each context change will always pass through N operators to reach an implication. For the average case, the shape of an antecedent will be a tree of Operators (AND or OR) with a branching of two. The number of contexts used in one implication is $2^{(N-1)}$. As each AND need all its sub-expressions to be active and each OR only one, most of the context changes should never reach the implication and so return sooner. The best case being stopping on the first Operator and the worst being reaching an implication.

The tests are run as follows: All the contexts are activated in order one by one in a transaction. They are then all deactivated one by one as for the activations. This is then repeated several times.

The average time for a given value is always calculated on the 202400 transactions (1000 times the loop explained early). A warm-up period of 20240 transactions is run before.

The results are shown in the figure 6.4.

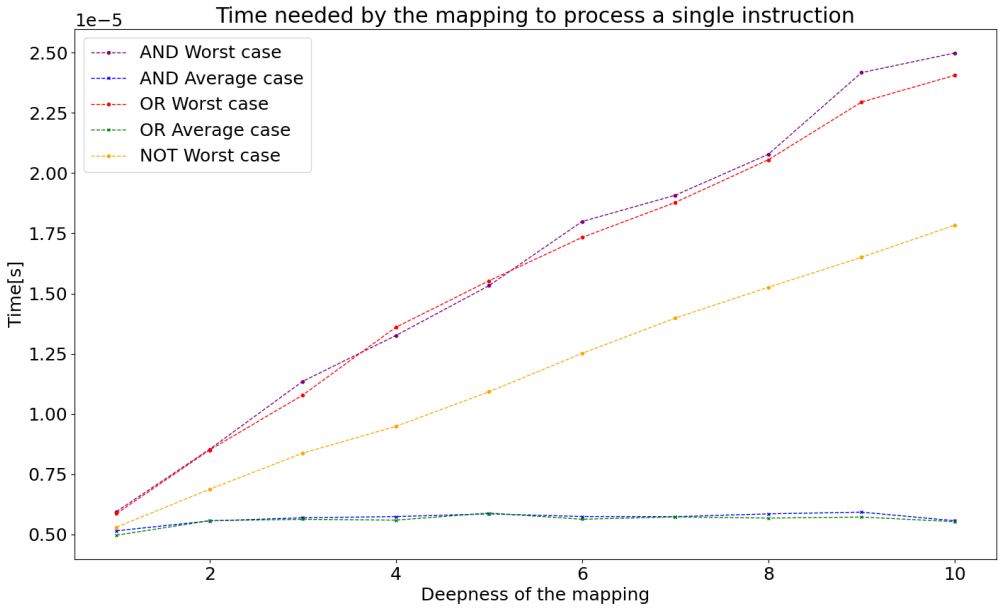


Figure 6.4: Influence of the deepness of the mapping on the average time needed to treat an instruction

This show that the time complexity is linearly dependent on the deepness of the mapping in the worst case. But on average, as the mapping is a tree, the time complexity will be in $\Theta(1)$. This average time complexity is also valid when the tree is unbalanced. Indeed, if an unbalanced tree has several children that are balanced trees. As all the balanced trees have the same average complexity, the unbalanced tree created with these trees shares the same average complexity, $\Theta(1)$. So recursively, it proves that any unbalanced trees has the same average complexity that a perfectly balanced tree.

The use of the Not expression should always be coupled with a context or another expression as the negation of the negation do not make any sense. In consequence, in the worst case all expressions (which are not a Not) could have a negation. But even in this case, the complexity will not change as it just multiply it with a constant factor: 2.

Summary

To summarize, the first experiment, with the model size as the factor, shows that the new model has a time complexity independent of the size of the model. Whereas the second one, with the deepness of the mapping as the factor, shows that the average time complexity is independent of the mapping deepness. Together, this means that in practice the time of a transaction depends only on the number of

context changes.

Another advantage is that no matter how the mapping is declared, its average time complexity stays the same assuming that the developer does not use the And or Or operator with a single child or use a Not on another Not. This allows splitting the expressions in several sub-expressions for the clarity of the mapping declaration without degrading the performance of the model.

But even if the new model is very fast, this advantage is moderate. In most COP applications, context changes do not happen often and therefore transactions either. In consequence, the time gained in the model is probably not noticeable. But for real-time applications with fast context changes, this improvement is really appreciable.

6.4 Conclusion

The new implementation allows having a more expressive and flexible mapping declaration without degrading the performance of the models. The introduction of the OR and NOT operators bring the new expressiveness. The possibility to split an expression in sub-expression and to reuse them bring the flexibility needed to build large applications. Overall, the time complexity of the new implementation is independent of the size of the models. And its average complexity is independent of the mapping declaration. So, the average time needed by the model only depends on the number of changes requested by the developer. The new implementation can so support very large applications with complex mapping without degrading its performance.

Chapter 7

Language improvement

This chapter will be about the improvements made to the feature execution and the proceed mechanism. In a first time, the old and the new architecture will be presented with the motivation to switch from the original to the new one. Then, the new implementation of these elements will be detailed. Next, the performances of this new implementation will be evaluated for the transactions and the global execution.

7.1 Architectures comparison

This section first presents the principle behind the Feature Execution in FBCOP. Then the method used to implement it in the original implementation is given followed by the one proposed in this master's thesis. These two implementations are then compared.

7.1.1 Principle

In FBCOP, the developer declares a skeleton and the behavioral variations of its application. The skeleton is composed of classes that can be empty or with functions considered as its default behavior. A behavioral variation is one or several modules composed of functions that can adapt a class. In the feature declaration, the developer links, inside a feature, the behavioral variations to one or several classes which creates the adaptations.

When a transaction activates or deactivates a feature, the corresponding feature-action is transmitted to the Feature Execution. Each feature-action is processed as follows. First, the adaptations linked to this feature is found. Then if it is an activation, the adaptations is added on the stack of their class. If it is a deactivation, the adaptations are removed from their stacks.

When a function is called in a class, the function with the same name from the last activated adaptation (which implement it) is called. A proceed in this function calls the previous activated version, and so on.

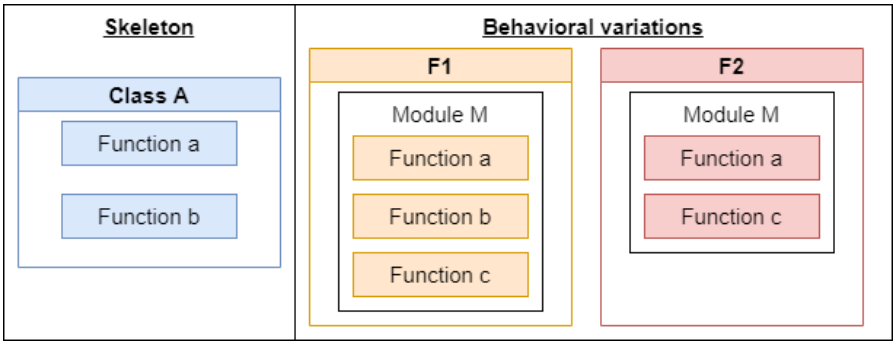


Figure 7.1: Declaration of the Skeleton and the Behavioral Variations

For example, let's take the declaration presented in the figure 7.1. There are two behavioral declarations (F1 and F2). In the feature declaration, two features of the same names adapt the class a. This creates one adaptation for each behavioral adaptation that can be used on this class.

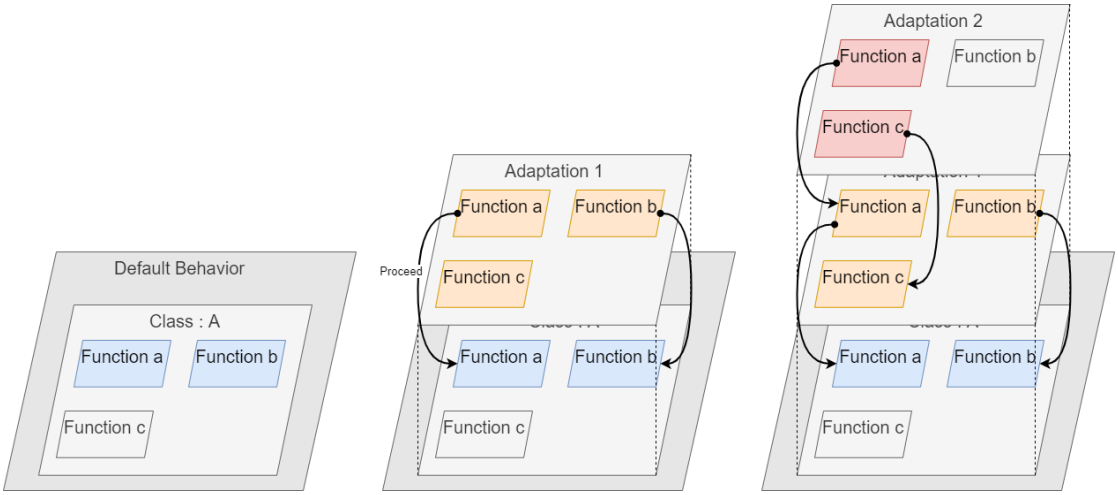


Figure 7.2: Left to Right, the default state , the state after the activation of F1, the state after the activation of F1 and then F2

The left of the figure 7.2 shows the default state of the adaptations. The function in gray can not be called as it is not implemented by default.

If the feature execution receive the feature action "*activate F1*", it finds the Adaptation 1 and activates it. The resulting state is in the center of the figure 7.2. If then it receives "*activate F2*", the final state of the adaptation is as shown on the right of figure 7.2.

If the "*function a*" is called at this moment, the function of the adaptation 2 (in red) is called first, its proceed calls the "*function a*" of the adaptation 1 (in orange)

where the proceed calls the default function (in blue). The proceed are shown in the figure 7.2 with the arrows.

7.1.2 Current method

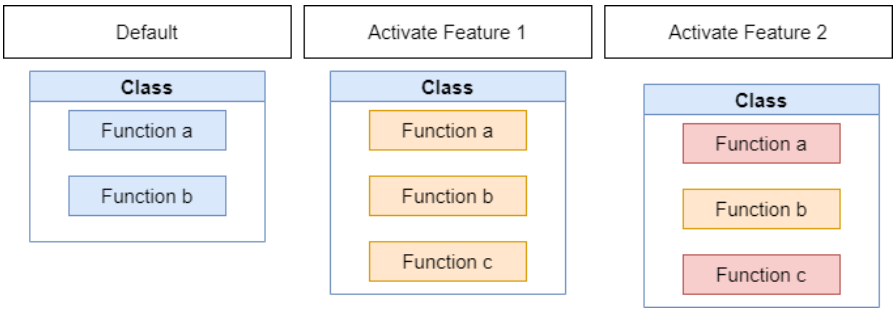


Figure 7.3: The resulting states of the current method used for the adaptation activation

The current method for the activation of an adaptation is to replace the function already deployed with the one of the new adaptation. The previous one is stored in a stack. The deactivation of the adaptation removes its function from the class or from the stack. If the function was the last deployed, it is removed from the class, as it is deployed, and replaced by the one on the top of the stack.

The figure 7.3 represents the states of the class according to the already used example presented in figure 7.2.

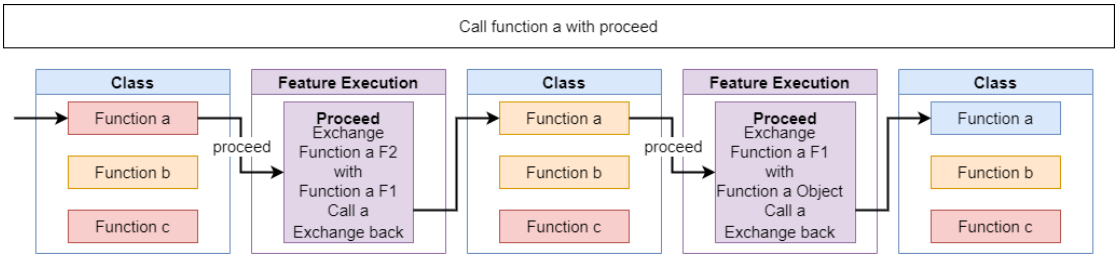


Figure 7.4: The current method used in function call

The figure 7.4 shows how the call of a function works. The call of the "*function a*" directly calls its last deployed adaptation (in red). As only one version of a function is deployed at a time, the proceed replaces the function with the last on the stack (in orange) and then calls it. The proceed of this new function does the same with the default function (in blue). When the orange function ends, the proceed restores the states of the class and then returns the value returned by the orange "*function a*".

7.1.3 Proposed method

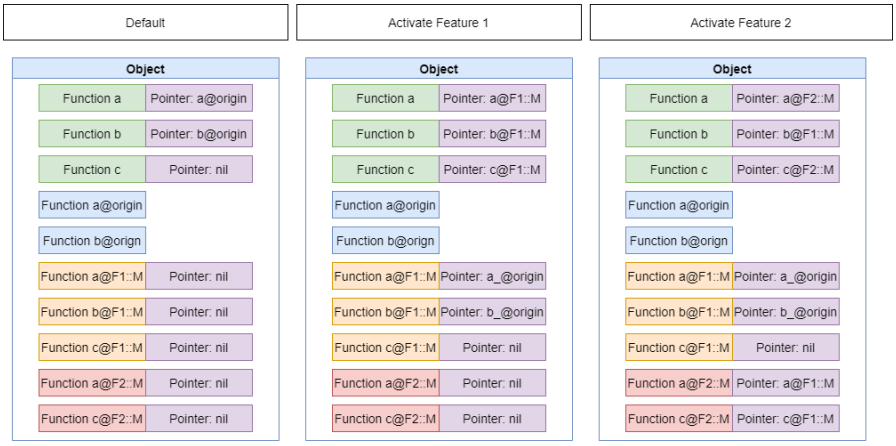


Figure 7.5: The resulting states of the proposed method used for the adaptation activation

The proposal is to use pointers instead of replacing the functions. In consequence, all the methods are deployed all the time but they are reachable only when the adaptation is active. A similar approach is used in subjectiveC [8].

All the possible callable function names, from the skeleton or an adaptation, are created with a function that only call a pointer (in green on figure 7.5). If the pointer is not nil, the function is called and if not it raises an error. The function of the skeleton (in blue) as all the functions of the adaptation are renamed by adding a tag to their names and added to the class (in orange or red). Each function that comes from an adaptation also has a pointer that indicated which function the proceed must call. The activation or deactivation of an adaptation simply changes the pointers to add or remove its functions.

The figure 7.5 shows the default state of a program on left. The "*function a and b*" point to their default implementations where the function c has no current implementation. The state of the pointer after the activation of F1 and F2 are also shown as in the figure 7.5 respectively on the middle and on the right.

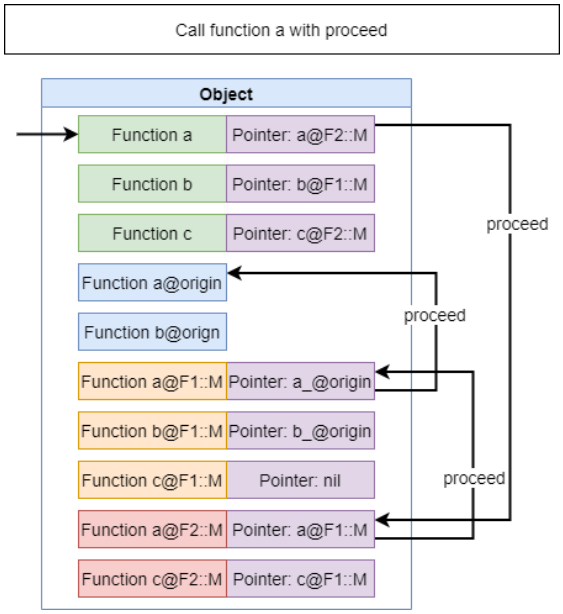


Figure 7.6: The proposed method used in function call

The call of the "*function a*" with F1 and F2 active is shown in figure 7.6. The "*function a*" calls the implementation of the adaptation 2 (in red) where its pointer calls the function of the adaptation 1 (in orange). The proceed of this last function calls the default implementation (in blue).

7.1.4 Comparison

Both methods have advantages and disadvantages.

The current method has the main advantages to not add a time overhead when a function is called. But it has two disadvantages. The first is that the proceed is slow as it must replace the function in the class. The second is that it is not compatible with multithreading as replacing the functions affect all instances of the class in all the threads.

The proposal is its exact opposite. The first advantage is it allows making multithreading as the state of a class do not change during the proceed. The second is that the proceed is very fast as it only consists to call a pointer. But the disadvantage is that the call of any function add a small time overhead.

The use of the proposal is based on two main arguments.

The main hypothesis is that any cop application should use the proceed in nearly all the functions. And so, if the time overhead is small enough, the performance of any application will be improved with the proposal.

The second is that multithreading is important for the applications of the COP paradigm. The languages are mainly used for graphical applications that should be multithreaded with at least one thread for the graphical part and one for the logic. The second possible application is embedded applications that should run in real time and so use multithreading to handle the different tasks.

7.2 Implementation

The implementation of the adaptation mechanism should be inserted in the default function call mechanism of the language. But Ruby not allows modifying this mechanism itself thanks meta programming. In consequence, the pointers and the functions can not be directly linked together. An external structure of data is so used to keep the pointers associated with the functions. This data structure is simply a singleton object with a hash map of hash map. The first hash map has as key the name of a Class and as value a hash map. This second hash map has as key the name of a function and as value an array with the previous function and the next function of the proceed order. The pointer on the previous function is necessary for the deactivation of the function. A second structure of data is used to keep which feature has which alterations with which functions.

7.2.1 Adaptations Initialization

The first thing done in the initialization is to change the name of the functions already implemented in the classes. These functions are renamed with their name extended with "@original". A function with the initial name is then re-created. This default function has the only vocation to redirect the calls on the function refereed by its pointer. This pointer is created in the pointer data structure and refer to the original implementation.

Then, the initialization finds all the features in the feature declaration that adapt a class. For each of them, it creates the adaptations and adds the functions inside the class with the name of the function extended with "@NameOfTheFeature::NameOfTheModule". If a function with the default name is not already in the class, this function is added and can only call the function refereed in the pointer data structure. This pointer is created and set to nil as no default implementation exist for this function. The function of the adaptation are also added to the pointer data structure with its new name and its pointer set to nil. The data structure that keep the functions of the alteration for a given feature is also updated.

The state after the initialization of the example of 7.2 is shown on the right of the figure 7.5. Only the next pointers are represented in this figure.

7.2.2 Feature Execution

The feature execution convert the feature name it receives into the functions of the adaptations. It then applies the action of the feature on these functions.

The activation of a function add this function on the stack used for the proceed as describe in the Listing 7.1. Thanks this sequence, the chain of pointer used in the execution is never broken as the update of a single pointer is atomic.

Listing 7.1: Activation of a function

```
def activate(class_name, function_name, name)
  function = get_pointers(class_name, function_name, name)
  default_function = get_pointers(class_name, function_name,
     function_name)

  function.previous = function_name
  function.next = default_function.next
  default_function.next = name
  if function.next != nil then
    next_function = get_pointers(class_name, function_name,
       function.next)
    next_function.previous = name
  end
end
```

The deactivation of a function removes the function from the stack as the Listing 7.2 shows it. The next pointer of the deactivated function is not modified. Like that if a part of the program is in this function, it can continue its execution without error.

Listing 7.2: Deactivation of a function

```
def deactivate(class_name, function_name, name)
  function = get_pointers(class_name, function_name, name)
  previous_function = get_pointers(class_name, function_name,
      function.previous)
  previous_function.next = function.next
  if function.next != nil then
    next_function = get_pointers(class_name, function_name,
       function.next)
    next_function.previous = function.previous
  end
  function.previous = nil
end
```

7.2.3 Function call & Proceed

Any function call is handled by its default function. If the pointer of this function is not nil, it calls the pointed function. In the other case, the default function tries to call the *super* function. If this function not exist, it raises an error. The call of the super allows to not raise an error when a class does not have the implementation of a function but one of its parent classes has it.

The proceed is similar except it must first discover which function call it. It is done with the *caller_locations* function provide by Ruby. After that it simply call the function of its next pointer or raise an error if the pointer is nil.

7.3 Feature Execution Performance

The Feature Execution Performances is evaluated in function of the number of actions-features submitted to it. The comparison between the original and the new implementation and between the activations and deactivations of adaptations is also evaluated.

All the values are an average time made on a thousand measures. Of course at the start of the test, a warm-up period is let. The measures taken for the average are made after. It consists of measuring the time needed by the Feature Execution to complete a list of feature changes of a specific size. This list can be made of activations or deactivations but not both of it. The case with the activations and the one with the deactivation are not mixed in the averages.

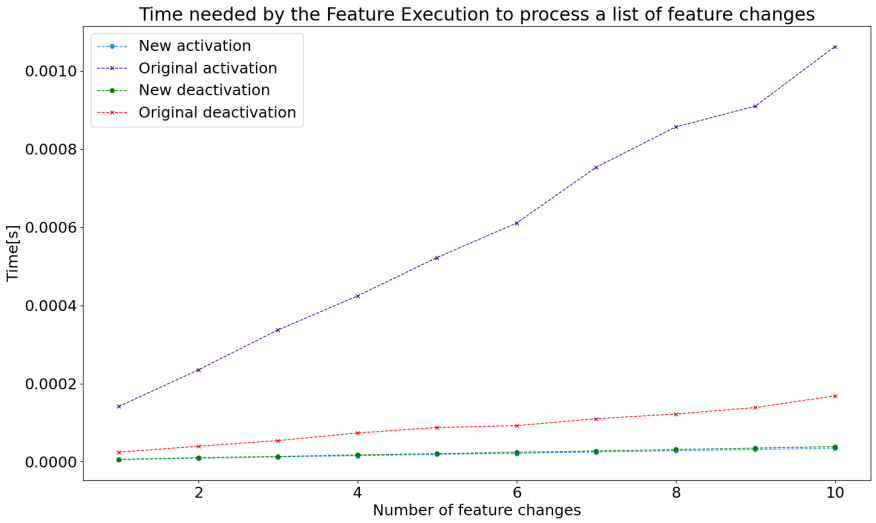


Figure 7.7: Comparison of the time needed by the Feature Execution to process a list of actions-features between the new and the original implementations.

The figure 7.7 shows that the feature execution depends on the number of feature changes in all the cases. But the new implementation grows very slowly when the original one grows faster. The deactivation or activation in the new implementation are similar and confounded in the graph. The original implementation is a lot faster in its deactivations than in its activations. But even the deactivation of the original implementation is slower than the activations or deactivations in the new implementation.

These results show clearly than the new implementation of the Feature Execution is more efficient than the original one.

7.4 Execution Performance

The comparison of the time needed to run a function is evaluated between the new and the original implementation.

This test is done with an empty function in the original class and with a function with only a proceed in the behavior adaptations. The time needed to return a function is evaluated on an average of a thousand calls. As in the other test, a warm-up period is let at the start of the test.

The times measured represent the overhead times that Ruby and the FBCOP implementation add to a function call with a given number of proceed.

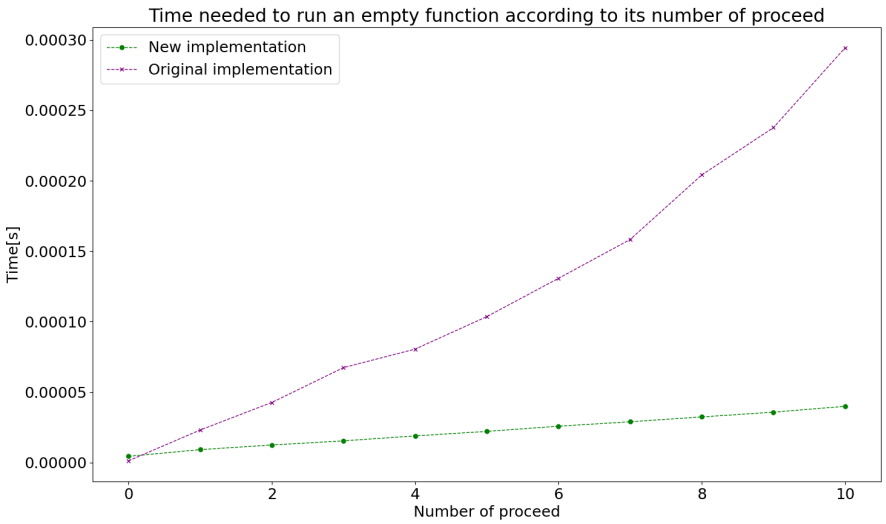


Figure 7.8: Comparison between the new and the original implementations of the time needed to run an empty function in function of its number of proceed

The figure 7.8 shows, as predicted, that the call of a function without proceed is faster in the original implementation than in the new one. But even with a single

proceed, the overhead added by the original implementation becomes bigger than the one of the new implementation.

As in FBCOP applications most of the functions use a proceed [1], the new implementation makes the execution of most FBCOP program faster.

7.5 Conclusion

A FBCOP application is composed of a skeleton of classes and of behavioral variations that can adapt these classes. The original implementation only deployed an adaptation at a time and store the other ones in a stack. So to pass from one to another, the adaptations are exchanged. This master's thesis proposed to use pointers instead. This gives the possibility to use FBCOP in multi-threaded applications. This new implementation also makes the Feature Execution step faster. On runtime execution, a slight time overhead makes the call of a function slower in the new implementation than in the original one. But as the proceed is faster in the new implementation and it is used nearly in all functions of a FBCOP applications, the overall execution is faster with the new implementation than with the original one.

[1] For example, in the COP Car application of the section 6.1 all functions, excepts a very few exceptions, use at least one proceed.

Chapter 8

Future works

This chapter gives some possible future improvements for FBCOP or its use. Firstly, about its model and then about the multithreading.

8.1 Model

This master's thesis mainly focuses on the model of FBCOP. Two possible improvements for the models could be made. The first is for the contexts. And the second a consequence of the new implementation which no longer works with the existing tools made to visualize the models.

8.1.1 Context with Data

In the original and the new implementation, the context can only have two states: active or inactive. But in some circumstances, this is difficult to represent a context as it is not boolean. This problem has been risen in the COP Car application where a context represents the speed of the car. Add the possibility to introduce data inside the contexts, as the speed of the car, could solve this problem and make a valuable addiction to FBCOP. Of course, the associate operators should be given to the mapping to handle these data.

8.1.2 Visualizer

The current implementation of FBCOP offers the opportunity to the developer to see the contexts, mapping, features and adaptations dynamically during the execution. This is done with two tools [6, 5]. But as the mapping presented in this master's thesis is different from the original one, these tools are not usable with this implementation. The tools could be updated to represent the new mapping with all its gates. This should help the developer to deals with bugs or to understand how the language work.

8.2 Multi-threading

Other COP languages are compliant with multi-threaded applications as EventCJ [15] or ContextJS [17]. But these languages do not share the layers between the

threads. In ContextJS the layers are thread specific and in EventCJ they are Object specific. This avoids a major part of the problems that a multithreaded system can have due to COP. In the new implementation of FBCOP, all the threads share the same contexts, features and adaptations. The consequence is that a FBCOP multithreaded applications must be implemented to respect the adaptation changes due to other threads. Some guideline should be defined to help the developer to build this kind of applications.

The new implementation allows multi-threaded applications to work without bugs as for the COP Car application. However, the multithreading was not tested deeply as it was far from the purpose of this master's thesis. Even if it works in most cases, there is no guarantee that there is no major bug of conception in some specific cases. An in-depth study would therefore be expected.

Chapter 9
Conclusion

This master's thesis shows that it is possible to improve the Feature-Based Context-Oriented Programming language. This language uses three models, that model the contexts, the features and the mapping which links them together.

A new declaration of this mapping is proposed and bring more flexibility and expressiveness. The new expressiveness is introduced thanks to the introduction of the Or and Not operators in addition to the And. The possibility to define and reuse sub-expression in the mapping declaration brings the readability and flexibility needed for such statements. Together, it makes possible to use the mapping declaration in large and complex applications.

The new implementation of the three models in once, proposed in this work, not only gives the possibility to use the new mapping declaration. It also optimizes the entire model and make its time complexity independent of the size of the model. Furthermore, thanks to the use of gates in the mapping implementation, the average complexity of the mapping is independent of its shape. So, no matter how a developer declares the mapping, its performances will always be the same.

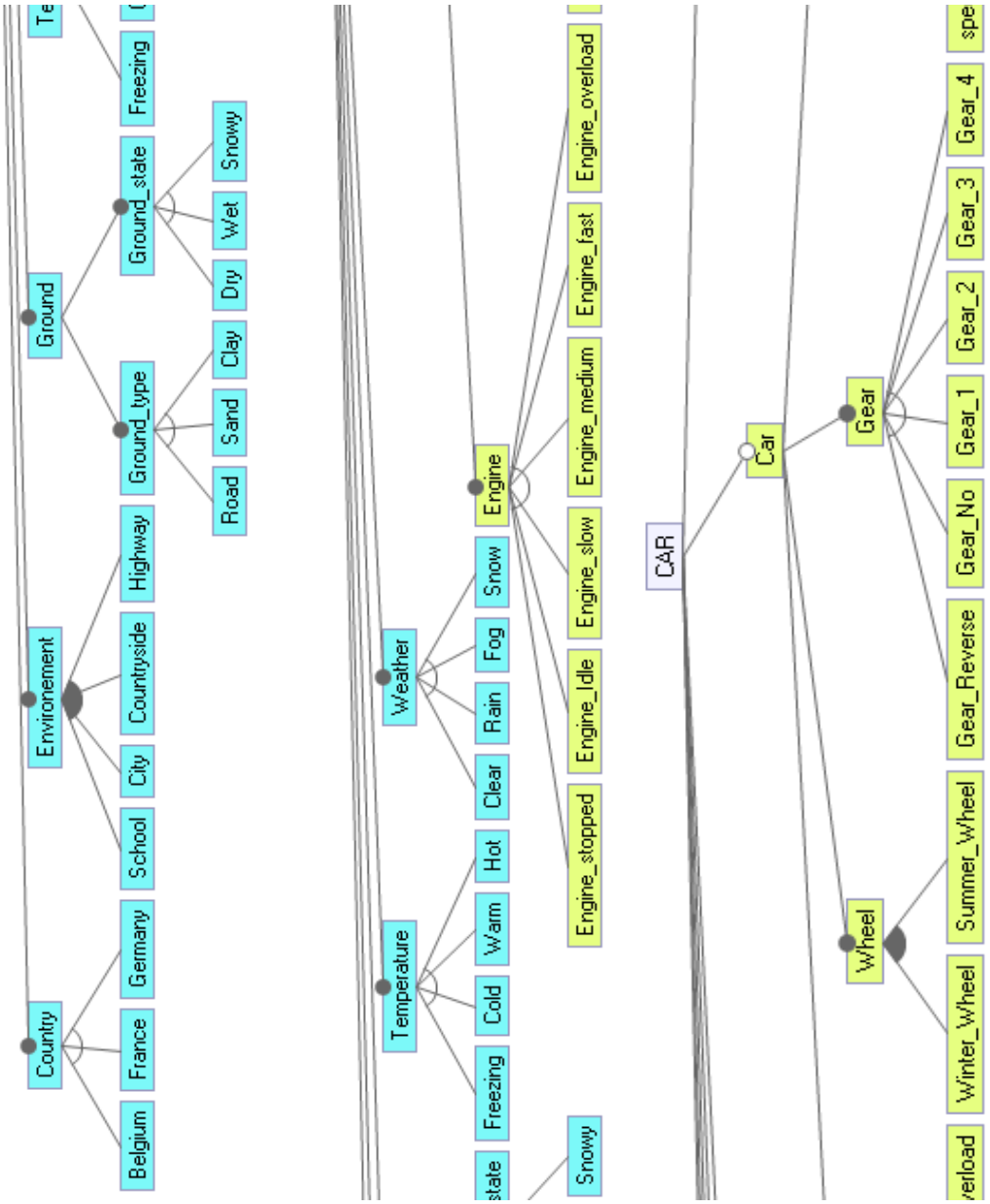
This work also introduces the notion of transaction to perform the context changes. Four different strategies to make a transaction are proposed. To evaluate these strategies, six properties were found. According to this evaluation, three of these strategies have an interesting behavior. So the strategy used for a transaction must be chosen between them according to the application. The last one do not add any new behavior but has more disadvantages, and so is not recommended.

A new implementation of the Adaptations mechanism is also proposed in this master's thesis. It uses pointers rather than replacing the function during the execution. This new version is compliant with multithreading whereas the original version was not. Moreover it also increases the performance during the change of behavior in the Feature Execution but also make the overall execution faster.

This work shows that it should be possible to make an implementation of FBCOP usable in any large and complex application.

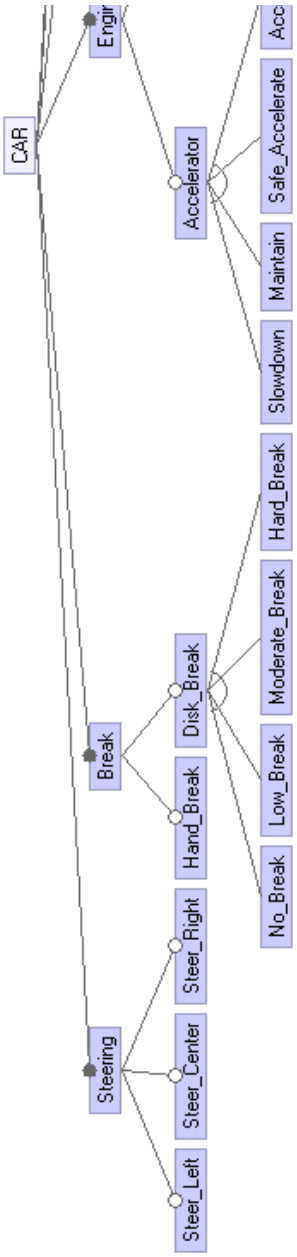
Appendices

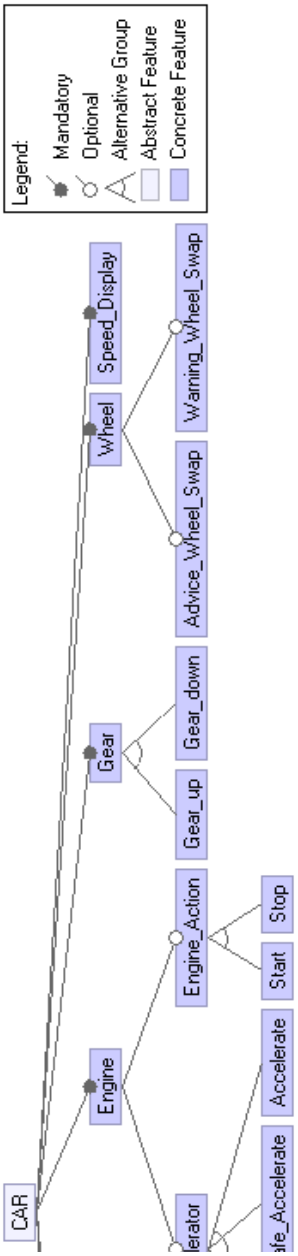
Appendix A
Car Contexts



Speed
speed_0
speed_0-10
speed_10-30
speed_30-50
speed_50-70
speed_70-90
speed_90-120
Gear_4
Steering
Steering_Left
Steering_center
Steering_Right
Direction
speed_upper_130
speed_120-130
Driver
Mode
Pedal
Sport
Break
Gas
GasSmall
GasFull
Mode_Forward
Mode_Stop
Mode_Reverse
Steering_Left
Legend:
Mandatory
Optional
Or Group
Alternative Group
Abstract Feature
Concrete Feature

Appendix B
Car Features





Appendix C
Car Mapping Declaration

```
require_relative '../../src/interface/mapping/expressions.rb'

module MappingDeclaration
  def self.create()
    #steering (easy)
    steer_left = 'SteeringLeft'
    steer_center = 'SteeringCenter'
    steer_right = 'SteeringRight'

    rule_steer_left   = Implication.new(steer_left, ['
       ↪ DisplaySteerLeft'])
    rule_steer_center = Implication.new(steer_center, ['
       ↪ DisplaySteerCenter'])
    rule_steer_right  = Implication.new(steer_right, ['
       ↪ DisplaySteerRight'])
    rules_steering = [rule_steer_left, rule_steer_center,
       ↪ rule_steer_right]

    #general
    off_road = Not.new('Road')
    sport = 'Sport'
    no_sport = Not.new(sport)

    no_vision    = Or.new('Rain', 'Fog', 'Snow')
    not_freezing = Not.new('Freezing')

    ice       = And.new('Wet', 'Freezing')
    mud       = And.new('Clay', 'Wet', not_freezing)
    dry_sand  = And.new('Dry', 'Sand')
    icy_road  = And.new('Road', ice)
    wet_road  = And.new('Road', 'Wet')

    very_slippy = Or.new('Snowy', dry_sand, mud, icy_road)
    slippy      = Or.new(wet_road, off_road, very_slippy)
    grippy      = Not.new(slippy)
```

```
#speed
stopped      = 'Speed0'
moving       = Not.new(stopped)
faster_130 = 'SpeedUpper130'
faster_120 = Or.new(faster_130, 'Speed120-130')
faster_90  = Or.new(faster_120, 'Speed90-120')
faster_70  = Or.new(faster_90, 'Speed70-90')
faster_50  = Or.new(faster_70, 'Speed50-70')
faster_30  = Or.new(faster_50, 'Speed30-50')

#unrespected_mode
not_reverse = And.new(moving, 'Front', 'ModeReverse')
not_stopped = And.new(moving, 'ModeStop')
not_forward = And.new(moving, 'Back', 'ModeForward')
unrespected_mode = Or.new(not_reverse, not_stopped,
    ↪ not_forward)

#breaking
hand_break      = And.new('ModeStop', stopped)
active_break    = Or.new('Break', unrespected_mode)
no_break        = Not.new(active_break)
low_break       = And.new(active_break, very_slippy)
moderate_break = And.new(active_break, slippy, Not.new(
    ↪ very_slippy))
hard_break      = And.new(active_break, grippy)

rule_hand_break      = Implication.new(hand_break, ['
    ↪ DisplayHandBreak'])
rule_no_break        = Implication.new(no_break, ['
    ↪ DisplayNoBreak', 'SimulatorNoBreak'])
rule_low_break       = Implication.new(low_break, ['
    ↪ DisplayLowBreak', 'SimulatorLowBreak'])
rule_moderate_break = Implication.new(moderate_break, ['
    ↪ DisplayModerateBreak', 'SimulatorModerateBreak'])
rule_hard_break      = Implication.new(hard_break, ['
    ↪ DisplayHardBreak', 'SimulatorHardBreak'])
rules_breaking = [rule_hand_break, rule_no_break,
    ↪ rule_low_break, rule_moderate_break, rule_hard_break]

#speed_limitations
german_highway  = And.new('Germany', 'Highway')
france_highway  = And.new('France', 'Highway')
belgian_highway = And.new('Belgium', 'Highway')
```

```
difficult_road = Or.new(slippy, no_vision)
normal_road = Not.new(difficult_road)
highway_under_restriction = And.new(difficult_road, '
    ↪ Highway')
countryside_difficult = And.new(difficult_road, '
    ↪ Countryside')
countryside_normal = And.new(normal_road, 'Countryside')

unlimited_speed = Or.new(sport, german_highway, off_road)
limited_speed   = Not.new(unlimited_speed)

max_30  = And.new(limited_speed, 'School')
max_50  = And.new(limited_speed, 'City')
max_70  = And.new(limited_speed, countryside_difficult)
max_90  = And.new(limited_speed, Or.new(countryside_normal,
    ↪  highway_under_restriction))
max_120 = And.new(limited_speed, belgian_highway,
    ↪ normal_road)
max_130 = And.new(limited_speed, france_highway,
    ↪ normal_road)

unrespected_30  = And.new(faster_30, max_30)
unrespected_50  = And.new(faster_50, max_50)
unrespected_70  = And.new(faster_70, max_70)
unrespected_90  = And.new(faster_90, max_90)
unrespected_120 = And.new(faster_120, max_120)
unrespected_130 = And.new(faster_130, max_130)

unrespected_limitation = Or.new(unrespected_30,
    ↪ unrespected_50, unrespected_70, unrespected_90,
    ↪ unrespected_120, unrespected_130)

#accelerators
slowdown         = Or.new(active_break,
    ↪ unrespected_limitation, Not.new('Gas'))
not_slowdown     = Not.new(slowdown)
maintain         = And.new(not_slowdown, 'GasSmall')
full_gas         = And.new(difficult_road, no_sport)
safe_gas         = Or.new(normal_road, sport)
safe_accelerate = And.new(not_slowdown, 'GasFull', full_gas
    ↪ )
accelerate       = And.new(not_slowdown, 'GasFull', safe_gas
```

```
    ↪ )

rule_slowdown         = Implication.new(slowdown, ['
    ↪ DisplaySlowdown', 'SimulatorSlowdown'])
rule_maintain         = Implication.new(maintain, ['
    ↪ DisplayMaintain', 'SimulatorMaintain'])
rule_safe_accelerate = Implication.new(safe_accelerate, ['
    ↪ DisplaySafeAccelerate', 'SimulatorSafeAccelerate'])
rule_accelerate       = Implication.new(accelerate, ['
    ↪ SimulatorAccelerate', 'DisplayAccelerate'])
rules_accelerator = [rule_slowdown, rule_maintain,
    ↪ rule_safe_accelerate, rule_accelerate]

#gears
forward_gears = Or.new('Gear1', 'Gear2', 'Gear3', 'Gear4')
reverse_gear  = 'GearReverse'
no_gear       = 'GearNo'

expected_no_gear = Or.new(And.new(stopped, slowdown), And.
    ↪ new(active_break, 'Speed0-10'), 'EngineStopped', '
    ↪ ModeStop', unrespected_mode)
expected_gear    = Not.new(expected_no_gear)
expected_reverse = And.new(expected_gear, 'ModeReverse')
expected_forward = And.new(expected_gear, 'ModeForward')

unexpected_reverse = And.new(expected_no_gear, reverse_gear
    ↪ )
unexpected_forward = And.new(expected_no_gear,
    ↪ forward_gears)
no_gear_to_forward = And.new(not_slowdown, expected_forward
    ↪ , no_gear)
no_gear_to_reverse = And.new(not_slowdown, expected_reverse
    ↪ , no_gear)

forward              = And.new(expected_forward,
    ↪ forward_gears)
sport_gear_up        = And.new(forward, 'EngineOverload',
    ↪ not_slowdown)
sport_gear_down      = And.new(forward, sport, 'EngineSlow')
no_sport_gear_up     = And.new(forward, no_sport, 'EngineFast
    ↪ ', not_slowdown)
no_sport_gear_down = And.new(forward, 'EngineIdle')
forward_up           = Or.new(sport_gear_up, no_sport_gear_up
```

```
    ↪ )
forward_down         = Or.new(sport_gear_down,
    ↪ no_sport_gear_down)
forward_gear_up      = And.new(forward_up, Not.new('Gear4'))
forward_gear_down    = And.new(forward_down, Not.new('Gear1')
    ↪ )

gear_up    = Or.new(forward_gear_up, no_gear_to_forward,
    ↪ unexpected_reverse)
gear_down = Or.new(forward_gear_down, no_gear_to_reverse,
    ↪ unexpected_forward)

rule_gear_up    = Implication.new(gear_up, ['DisplayGearUp',
    ↪  'SimulatorGearUp'])
rule_gear_down = Implication.new(gear_down, ['
    ↪ DisplayGearDown', 'SimulatorGearDown'])
rules_gear = [rule_gear_up, rule_gear_down]

#wheel
winter_wheel_warm      = And.new('WinterWheel', 'Warm')
winter_wheel_hot       = And.new('WinterWheel', 'Warm')
summer_wheel_cold      = And.new('SummerWheel', 'Cold')
summer_wheel_freezing = And.new('SummerWheel', 'Freezing')

advice_wheel_swap  = Or.new(winter_wheel_warm,
    ↪ summer_wheel_cold)
warning_wheel_swap = Or.new(winter_wheel_hot,
    ↪ summer_wheel_freezing)

rule_advice_wheel_swap    = Implication.new(
    ↪ advice_wheel_swap, ['DisplayAdviceWheelSwap'])
rule_warning_wheel_swap = Implication.new(
    ↪ warning_wheel_swap, ['DisplayWarningWheelSwap'])
rules_wheel = [rule_advice_wheel_swap,
    ↪ rule_warning_wheel_swap]

#start stop

start_engine = And.new(Not.new('ModeStop'), 'EngineStopped
    ↪ ', 'GearNo')
stop_engine  = And.new('ModeStop', Not.new('EngineStopped')
    ↪ , 'GearNo')
```

```
    rule_start    = Implication.new(start_engine, ['DisplayStart
      ↪ ', 'SimulatorStart '])
    rule_stop     = Implication.new(stop_engine, ['DisplayStop ',
      ↪  'SimulatorStop '])
    rules_engine = [rule_start, rule_stop]
    #total
    return rules_steering + rules_breaking + rules_accelerator
      ↪ + rules_gear + rules_wheel + rules_engine
  end
end
```

Bibliography

[1] Rafael Capilla, Óscar Ortiz, and Mike Hinchey. Context variability for context-aware systems. *Computer*, 47:85–87, 02 2014.

[2] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, Jorge Vallejos, and Theo D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71–94, 10 2014.

[3] Pascal Costanza. Context-oriented programming in contextl: state of the art. *LISP50: Celebrating the 50th Anniversary of Lisp*, (4):1–5, October 2008.

[4] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. 01 2005.

[5] Benoît Duhoux, Bruno Dumas, Kim Mens, and Hoo Sing Leung. A context and feature visualisation tool for a feature-based context-oriented programming language. 12 2019.

[6] Benoît Duhoux, Kim Mens, and Bruno Dumas. Feature visualiser: an inspection tool for context-oriented programmers. pages 15–22, 07 2018.

[7] Benoît Duhoux, Kim Mens, and Bruno Dumas. Implementation of a feature-based context-oriented programming language. pages 9–16, 07 2019.

[8] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c: Bringing context to mobile platform programming. pages 246–265, 10 2010.

[9] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object system. *Journal of Universal Computer Science*, 14:3307–3332, 01 2008.

[10] Sebastián González, Kim Mens, Marius Coluacioiu, and Walter Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. pages 209–220, 03 2013.

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

[12] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with contexts. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2007.

[13] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[14] Zakwan Jaroucheh, Xiaodong Liu, and Sally Smith. Mapping features to context information: Supporting context variability for context-aware pervasive applications. volume 1, pages 611 – 614, 10 2010.

[15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: A context-oriented programming language with declarative event-based context transition. pages 253–264, 01 2011.

[16] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[17] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Sci. Comput. Program.*, 76:1194–1209, 12 2011.

[18] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Yu. Context aware reconfiguration in software product lines. pages 41–48, 01 2016.

[19] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. Modeling and managing context-aware systems' variability. *IEEE Software*, 34(6):58–63, 2017.

[20] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. 04 2016.

[21] Grzegorz Nalepa and Szymon Bobek. Rule-based solution for context-aware reasoning on mobile devices. *Computer Science and Information Systems*, 11:171–193, 01 2014.

[22] Sinisa Neskovic and Rade Matić. Context modeling based on feature models expressed as views on ontologies via mappings. *Computer Science and Information Systems*, 12:35–35, 08 2015.

[23] Thibault Poncelet and Loïc Vigneron. The phenomenal gem; putting features as a service on rails. Master's thesis, Université catholique de Louvain, 6 2012.

[24] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85:1801–1817, 08 2012.

[25] Sandra Trullemans, Lars Holsbeeke, and Beat Signer. The context modelling toolkit: A unified multi-layered context modelling approach. *Human-Computer Interaction*, 1, 06 2017.

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