UCLouvain

École polytechnique de Louvain

Strategies for managing the feature-interaction problems in context-oriented programming

Auteurs: **Benjamin Caudron, Arno Galand**
Promoteurs: **Kim Mens, Axel Legay, Benoît Duhoux**
Lecteur: **Xavier Gillard**
Année académique 2022–2023
Master [120] en sciences informatiques

Abstract

Context-oriented programs are able to dynamically change their behaviour depending on the environment they are currently executing in, however adding features to the paradigm can create new types of problems that we will call feature-interaction problems or more simply: conflicts.

In this thesis, we worked on the RubyCOP framework which implements the feature-based context-oriented programming paradigm FBCOP. This framework enables programmers to develop programs that can deploy new features, methods and classes at run time when the programs detect a context change during the execution. More precisely, we focused on problems that could occur during the execution of adaptive programs and how we could provide a solution to solve these problems. In fact, when two features are deployed together, they can enter into conflict with each other and create unexpected, unpredictable or faulty behaviours. Two main types of conflicts were defined:

- The first type is incompatibility conflict. It occurs when multiple incompatible features, that are activated together, lead to crashes or faulty behaviours. This type of conflict is hard to detect and requires some detection tools to do so.

- The second type is sequential conflict, this type of conflict occurs when the behaviour of the system depends on the feature activation order. A sequential conflict occurs when two executions with the same configuration (the same set of activated contexts and features) have different results.

Multiple solutions exist to handle those conflicts and one of them is using strategies. A strategy dictates how the system should behave in order to manage conflicts. The strategies used differ depending on the type of conflict, as the core problem is different. Incompatibility conflicts come from the interaction between features whereas sequential conflicts come from the feature activation order.

Strategies used to manage incompatibility conflicts are various and can range from throwing errors, letting the user handle conflicts, reverting to a non-conflicting state and activating incompatible features independently to prioritising important features.

To manage sequential conflicts, strategies to influence the feature activation order (feature precedence order) need to be defined. We will explore in this thesis strategies that use traversal algorithms such as DFS. But also precedence strategies that use either precedence scores or precedence relations between features.

Acknowledgments

Nous souhaitons par ces quelques lignes, remercier grandement les professeurs Kim Mens et Axel Legay ainsi que Benoît Duhoux qui nous ont permis de réaliser ce mémoire. Malgré notre mobilité au Japon et nos réunions d'un bout à l'autre du monde, votre aide et vos nombreux feedbacks nous ont été inestimables tout au long de ce travail. Votre pragmatisme et vos différentes expériences nous ont poussé à aller au-delà et nous ont donné le goût du chef-d'oeuvre.

Benjamin

J'aimerais tout d'abord remercier tous mes amis avec qui j'ai passé de magnifiques années ici à Louvain-la-Neuve mais aussi tous ceux que j'ai eu le plaisir de rencontrer ailleurs que ce soit ici en Belgique ou au Japon. J'aimerais également remercier ma famille qui m'a toujours soutenu dans tout ce que j'ai entrepris jusqu'à présent et notamment dans mes études. Je remercie également ma petite amie de toujours me soutenir au quotidien et de tous les bons moments qu'on passe ensemble. Je souhaite particulièrement remercier Arno avec qui j'ai pu réaliser cette thèse, ainsi que partager un Erasmus incroyable et des belles années d'amitié. Je remercie également nos promoteurs Kim, Axel et Benoît qui nous ont grandement aidés et encadrés pour ce mémoire, mais aussi tous mes professeurs, assistants et l'UCL de m'avoir tant appris. Et finalement, merci à tous ceux qui ont eu un impact positif dans ma vie, la liste serait trop longue pour tous les citer, mais je vous garde une place chaleureuse dans mon coeur.

Arno

C'est ainsi que toutes ces années à Louvain-la-Neuve se terminent. Ces années, je les ai passées avec des personnes formidables. Des amis qui auront une place privilégiée dans mon coeur. Qui, quand le jour pour moi était comme la nuit, étaient à mes côtés. Benjamin, merci pour ces si bons moments passés avec toi, ici en Belgique, comme au pays du soleil levant. Ta fougue de vivre et ta positivité sont d'une chaleur rayonnante. Maxime, à nos nombreuses balades en vélos et nos

réflexions philosophiques. Benoît et Christophe car derrière chaque étudiant, il y a de grands assistants. Merci à ma famille et mes parents. Merci à Gil, mon frère, qui est la lumière qui guide mes pas. Sans votre soutien et vos conseils, rien de tout cela n'aurait jamais été possible. Sachez que ma porte vous sera toujours ouverte.

Contents

Chapter 1

Introduction

1.1 Setting the scope

When talking about feature-interaction problems, we first need to know when and where it occurs. The context-oriented programming paradigm allows us to create software that reacts to the environment and will be able to deploy new features and new behaviours during their execution. FBCOP: Feature-based context-oriented programming, is one sub-class of the Context-oriented programming paradigm. It is a combination of feature modelling, a mapping model and the Context model (COP). RubyCOP is a framework developed mainly by Benoît Duhoux as his PhD thesis [9] and other students as their master thesis. RubyCOP is built on top of the Ruby programming language, available for developers to build adaptive software. Programs resulting from this framework will be able to deploy features dynamically depending on the context of execution. A mapping will link the features to deploy and the contexts. Such ability come with risk during the running time. When, for example, two features enter in conflict with each other.

Developers designing feature-based context-oriented software want to make sure that their programs are robust, fault-tolerant and working as expected. Designing such software requires being extremely cautious about both the big picture and smaller details, such as verifying the outcome of activating any set of features together. This verification is crucial, as many errors and unexpected outputs can arise later if it was not done carefully. Even when doing it carefully, which is a long and tedious process, some errors can remain undetectable or extremely hard to detect statically. These errors will be named 'conflicts' and, finding strategies to avoid them will be the main concern of this thesis.

Two types of conflicts will be discussed in this thesis. The sequential and the incompatibility conflicts.

- Incompatibility conflicts: In an adaptive system, when activating two contexts at run time, their features can be incompatible, leading to either crashes or wrong outcomes. This is an incompatibility conflict and it has to be detected by using detection tools in order to handle them by using strategies.

  To illustrate this, imagine a smart home system that activates sprinklers when sensing a fire and cuts off the water when sensing a flood. An example of incompatibility conflict is the result of sensing both the fire and the flood. The system does not know how to react and depending on the implementation of the system, this can lead to crashes or faulty behaviours.

- Sequential conflicts: In an adaptive system, for the same configuration (activated contexts, mapping and activated features), depending on the order of activation of the contexts, if two executions lead to different behaviours in the system, then, we have a sequential conflict. This type of conflict does not need a detection tool because the developer can check by himself if the behaviours are different.

  For example, we can imagine a smart application that changes the way it displays information depending on the context. If each time the user restarts the application, the display is different for the same context, it could be confusing for him. This is why we need to set the order of activation of features. This ensures that for each run, the same configuration will lead to the same execution.

Finding solutions to resolve such conflicts is crucial because only one of these conflicts is enough to break an entire program, resulting in faulty behaviours. Several solutions were already presented in [21] but no solutions to resolve those conflicts exist within the FBCOP approach. Creating solutions to handle conflicts and implementing them would enhance the FBCOP paradigm by making it less prone to conflicts and more consistent.

To handle these conflicts, strategies need to be implemented. A strategy is a tool that can be used by developers in order to resolve or avoid conflicts. Different sets of strategies exist depending on the type of conflict. One strategy used for an incompatibility conflict will not be the same as a strategy to resolve sequential conflicts. It is because both conflicts are intrinsically different problems and require different methods to handle them.

In this Thesis, we will explore those different conflicts and provide an implementation in the RubyCOP framework to help future developers use strategies for managing the context interaction problem.

1.2 Research questions

Several papers talk about feature-oriented programming but not many present the feature-interaction problems and conflicts that can occur in such adaptive systems. This thesis will therefore address the following research questions:

- **RQ1: What types of conflicts can occur in adaptive systems?**
  We already answered this question in the introduction. Indeed we defined two types of conflicts that can occur in feature-oriented systems: Incompatibility and sequential conflicts. We will describe those conflicts in detail and present concrete examples in their respective chapters 3 and 4.

- **RQ2: What are the possible solutions to handle or avoid those problems?**
  Strategies are our core solution to handle conflicts. While they were quickly mentioned in the introduction, the different strategies for each conflict will be presented and explained in their respective chapters 3 and 4.

- **RQ3: How can we implement those solutions so that developers can use them to handle or avoid those problems?**
  First, we need to know which strategies are possible to resolve conflicts. But then, we also need to implement those strategies in order to use them in concrete examples. We will look at how the implementation of the strategies can be done to resolve conflicts.

- **RQ4: How to use these solutions in practice to handle those conflicts depending on the type of program?**
  After implementing the strategies, we will try to compare them and look at different case studies to see how and when developers should use the different strategies.

1.3 Contributions

In this section, we will present the different contributions of this thesis to the FCBOP approach and the feature-interaction problems. Those contributions answer the research questions presented in the previous section.

- **C1: Precise definitions of the feature-interaction problems**
  Defining the feature-interaction problems means explaining in detail both incompatibility and sequential conflicts. This also includes explaining concrete examples, case studies, when it happens and how it can be detected. This contribution answers the research question **RQ1**.

- **C2: Explanations of several strategies to handle both types of conflict**
  Having multiple strategies allow more adaptability and the possibility to choose the right strategy depending on the type of program. This contribution addresses the research question **RQ2**.

- **C3: Implementation of several strategies to help developers resolve sequential conflicts**
  In order to give the tools to the developers to resolve sequential conflicts, we developed 3 strategies that are fully functional and that can be used in RubyCOP to remove all sequential conflicts. These tools can also help users design the layout of their RubyCOP apps as it gives them the opportunity to choose their activation order. By changing the feature activation order, developers can change the layout without having to code anything else. This contribution addresses the research question **RQ3**.

- **C4: Validation of the strategies to resolve sequential conflicts**
  Validation is necessary for any piece of work to ensure its validity and its correctness. Our validation compares the strategies together in different contexts by using our running example as a case study. This allows us to rank them compared to the type of program. This contribution addresses the research question **RQ4**.

1.4 Roadmap

We first presented the introduction of our thesis in this chapter (1) by hovering over the key concepts notably by defining both incompatibility and sequential conflicts. And we also presented our motivations, research questions and contributions.

Following that, we will show the state-of-the-art chapter (2) by talking about the background material and related work. This will help us find leads on what kind of strategies we can define to resolve both types of conflicts.

Then the next two chapters (3-4) will focus on incompatibility and sequential conflicts respectively. In those chapters, we will explain the conflict more in detail

and present the different strategies we discovered. The incompatibility chapter is also self-contained, meaning that it will also contain case studies and an analysis of the strategies.

The two following chapters (5-6) focus only on sequential conflicts by talking about the implementation and the validation respectively.

The implementation chapter (5) will present the strategies we implemented and how we did it. It also presents mechanisms and algorithms that were needed to implement them. Finally, it proves the time complexity of the algorithms.

The validation chapter (6) will validate and analyse the sequential conflicts, the implemented strategies and present case studies. In the end, it will compare the different strategies to find out which one is the most effective for different types of programs.

The two last chapters (7) will first The future work chapter will showcase our ideas to continue improving the FBCOP approach and more precisely the feature-interaction problem. This includes new strategies and mechanisms to detect and resolve both types of conflicts. Finally, chapter (8) will conclude the thesis. We will summarise our findings and present our final solutions to resolve conflicts.

Chapter 2

State of the art

This chapter is split into two parts. In the first part, we will explore the background material that forms the foundations of this thesis. We will give an overview of the COP paradigm followed by an explanation of the FBCOP paradigm.

In the second part, We will take a look at related work on the feature-interaction problem and the two types of conflicts we have seen in the introduction chapter. We will explain variations of this problem in COP and other programming languages, which induce our implementation choices.

2.1 Background Material

Context-oriented programming

Contextual information is a valuable resource with many different usages. There is an idea of using this information to create programs that will then be aware of, and adapt their behaviours to their context of execution. An emerging way of programming is specially dedicated to those programs: Context-Oriented-Programming. This paradigm is explored in the paper Context-oriented Programming by Robert Hirschfeld and al [11]. As explained in this work, context can represent a wide range of information: Real World data sensed through sensors, hardware information, and software information. With this large amount of data, COP programs have a lot of applications such as software evolution, and execution context dependencies. The main Motivation of COP is to enable programmers to write programs that adapt themselves depending on the context.

Feature-based context-oriented programming

In this thesis, we will use RubyCOP, a framework mainly developed by Benoît Duhoux [9]. This framework is specially dedicated to the COP approach. This tool implements a branch of the COP paradigm, Feature-Base Context-Oriented Programming, FBCOP [8]. The RubyCOP is a framework built on top of the Ruby programming language. It is specially designed to create programs that follow the FBCOP approach. Developers can explicitly declare feature, context and mapping models Using RubyCOP. Models (the features and the contexts) will be represented both by a tree-like architecture while the mapping will be used for the feature selection. As we will explain later, those trees include all the constraints the developers might have declared.
To summarise, the RubyCOP framework helps developers to create and model easily FBCOP programs.

Figure 2.1 summarize the architecture of FBCOP detailed in the work of Duhoux and al [8].

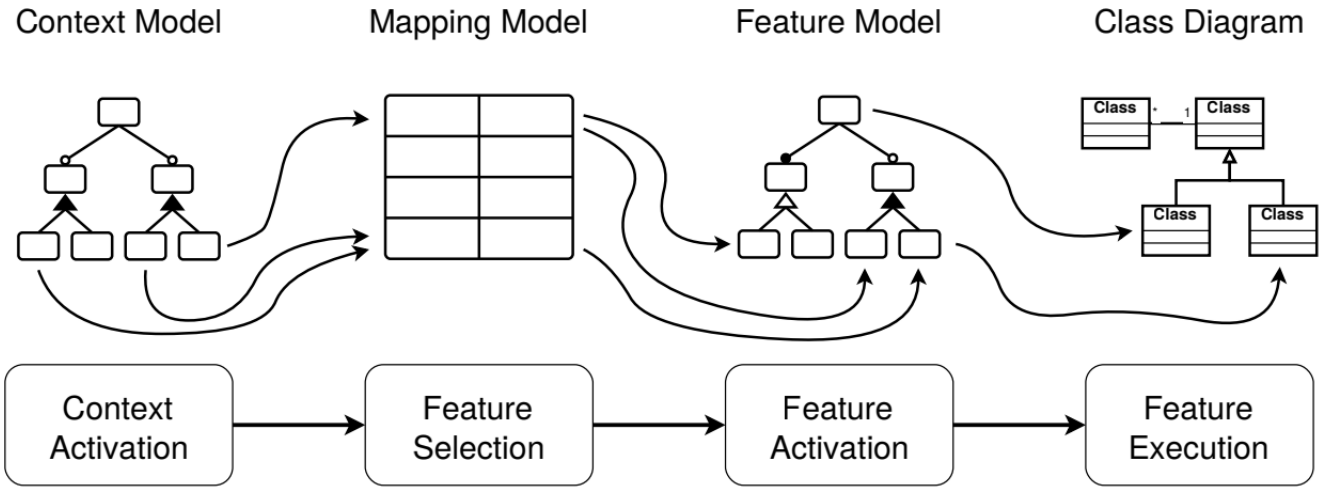

Figure 2.1: Overview of the FBCOP's system architecture [8]

When a change of context is sensed by the system, the corresponding context will be activated in the context model. Following the activation, the feature mapped to the context in the Mapping Model will be activated. When a feature in the feature model is activated, it is deployed and executed in the adaptive system. Everything is done at run time, making the system aware of the context execution. In order to fully comprehend this design, we are going to explain, component by component the FBCOP architecture.

In FBCOP, a feature is defined as follows: "Features are implementation components, or parts thereof, that are visible and distinguishable to the end-user and which may be relevant depending on the particular user or usage context" [9] Most of the features are designed to be "fine-grained". It means that the features are interacting with only a few components of the system (methods or classes). It must be fine-grained because their purpose is to adapt the behaviour of the system at run time by refining the current behaviour. This property is also desirable for re-usability and maintainability. Unfortunately, not all the features can be fine-grained. Some features interact with multiple classes or methods. For example, a feature changing the display size of texts or objects must interact and adapt all the concerned classes. Those features are called "coarse-grained". All the features can be represented in a "Feature Model". The features and the features modelling representation is inspired by Kang and al FODA [14].

The purpose of this model is to graphically show the full capability of the adaptive system. This representation also enables a better understanding of all features implemented and is a medium of communication between end users and developers. The model will be represented as a feature diagram, following the structure of a tree. Each node will represent a feature. Some of those nodes are mandatory for the system to work properly and other non-mandatory features. Since the feature model follows a tree-like structure, the features respect a hierarchical order, meaning a parent node cannot be activated after its children. In parallel the model includes constraints between nodes. Those constraints are "Mandatory", "Optional", "alternative" and, "or". Those constraints only apply to nodes with a parent-child relationship. Other constraints are "cross-tree", they are constraints that are applied not only in parent-child relations but can be applied to all the nodes in the tree. Those constraints represent exclusion or requirements dependencies. For example, feature A can require feature B in order to function. The same goes for the exclusion constraints. All of those constraints need to be satisfied in order to have a valid set of features. If one of those constraints is not respected, then the configuration proposed is not respecting the feature model. Another important thing to add concerning features is that they can be either Abstract or Concrete. Abstract features have no real interaction or functionalities, they are used as interfaces that help developers to group functionalities together. On the other hand, concrete features will implement the actual functionalities of the system and are used for adapting the system's functionalities dynamically to changing contexts.

FBCOP must also take into consideration the context. The centrepiece of the COP paradigm is to adapt the execution of the system depending on the different sensed

contexts. FBCOP did the choice to split the contexts sensed by the system and the features to which it will correspond. The solution that is used is to structure the contexts in the same way as the features. Contexts will be modelled using a tree like-structure. This representation can include the same constraints as the feature model we already described. Designing the context diagram the same way as the feature model allows the developers to have a better understanding of the context. Following the convention described in many papers including Semantics for consistent activation in context-oriented systems by Mens and al [22]. The context diagram is decomposing the contexts thus, the developers have a better understanding of the global context of the adaptive system. It also simplifies the reusability of the model. Once the contexts have been modelled, you can reuse them in another application.

Finally, the context-feature mapping is what links the context model with the feature model. The purpose is to be able to activate or deactivate features at run time. The mapping allows FBCOP to manage multiple scenarios of execution. The mapping is represented as a set of relations which, for one or more contexts, will activate the corresponding feature(s). The mapping allows us to fuse all the models together and in a single one, the context-feature mapping model. Such model mapping will be seen later on in this thesis section 6.1.

2.2 Related Work

In this related work section, we are going to describe the different papers we used to create our strategies for both incompatibility and sequential conflicts described in the Introduction section 1. The focus will be first on the multiple inheritance problems [20, 23] because they can be directly compared to the problems of both conflicts. We will then talk about one solution for inheritance: linearization [1, 7, 12], and how it can be used to resolve our conflicts. We will talk about the C3 algorithm [1] and Topological Sort [13]. Then, we will explore other solutions and mechanisms that are allowing us to create precedence orders. This will help in the resolution of sequential conflicts. These mechanisms are implemented in Aspect-Oriented Programming (AOP) [17, 3, 16] and Prolog [19, 4]. Finally, we will explore solutions proposed for incompatibility conflicts in layer-based context-oriented languages such as reuse contracts [**empty citation**] and Modular layer activation [18].

Multiple inheritance problem

Both incompatibility and sequential conflicts can be compared to the multiple inheritance problem, also named the repeated inheritance problem or the diamond problem. When a class inherits the same method from two or more parent classes that themselves inherit from an ancestor class, that can create an ambiguity. Indeed, it can inherit the method from either parent class. All languages that allow multiple inheritance must deal with this ambiguity by using some kind of strategy.

Papers review the problems behind multiple inheritances either by naming it the diamond problem [20], or the problem of repeated inheritance [23].

A similar problem exists within sequential conflicts because there is also an ambiguity in the feature activation order. For the same configuration, we can have multiple outcomes possible which means that the result is ambiguous. By looking at solutions implemented to deal with the diamond problem, we can take inspiration from that in order to resolve sequential conflicts.

The resolution to the diamond problem can also help with incompatibility conflicts. When an incompatibility conflict occurs, it means that two features are activated but incompatible together. Similar to the diamond problem, when a feature inherits the same method from several parents, one solution is to inherit the method of one parent based on chosen factors such as for example: developer's choice or linearization.

Linearization

One solution to the diamond problem is linearization. This approach is explained in the work of K. Barrett [1], R. Ducournau [7] and F. Hivert [12]. Linearization in Object Oriented programming language is the act of choosing in which order the classes will be inherited in case of multiple inheritance. This is done by running an algorithm that takes the hierarchical structure of the classes and outputs the class precedence order. It is a list of the class in monotonic ordering from the most specific class to the least specific class. The monotonic property is necessary for the inheritance mechanism to work properly and incrementally.

Linearization is exactly what we need to resolve our sequential conflicts because we can activate the feature in an order similar to the one given by linearization. By implementing strategies similar to the linearization algorithms we will see next, we will be able to resolve sequential conflicts.

As stated by Barrett [1], "Typically, a linearization is computed by merging a

set of constraints or, equivalently, topologically sorting a relation on a graph, though other mechanisms are possible.". This idea of topological sort [13] will be reused when implementing our relative precedence strategy in section 4.

We first need to explain the class precedence graph. We can represent a class hierarchy by a Directed acyclic graph (DAG), where each vertex is a class and each edge is an inheritance relation. After doing this, it becomes visually easier to determine which classes are the most specific. When doing the linearization, it is mandatory that the class order respects the strict partial order of the precedence graph to ensure that the hierarchical structure is respected.

To give a concrete example of a linearization algorithm, we will talk about the C3 linearization algorithm. It is one of the most popular algorithms used for linearization. Its goal is to return the Method Resolution Order (MRO). In languages that allow multiple inheritances, it is necessary to know in which order the class should inherit methods from its superclass. This is precisely what the MRO is, it is a list containing the monotonic order in which the methods should be inherited. This order has to respect 3 conditions (the name C3 comes from the algorithm being consistent with 3 properties), to be monotonic, and to respect both the extended precedence graph and the local precedence order. The local precedence order is the direct order of the superclasses. The extended precedence graph is the hierarchical order of the class based on the local precedence order but with the added transitivity created by the local precedence.

The C3 algorithm works by merging sequences of classes and using a tie-breaker with ambiguous cases. The tie-breaker works by choosing the first superclass in the local precedence order. The merge function takes all linearization lists of its parents and a list of its parent as arguments. It takes the first head (first element of a list) that is not present in the tail (all elements except the head) of all other argument lists. Then, it adds it to the MRO and repeats those two last steps until obtaining the final MRO. Because the output is a linearization order, the duplicates in the MRO are discarded.

We create a simple example of how the C3 algorithm works in Figure 2.2. The notation L(A) means the linearization order (MRO) of A.

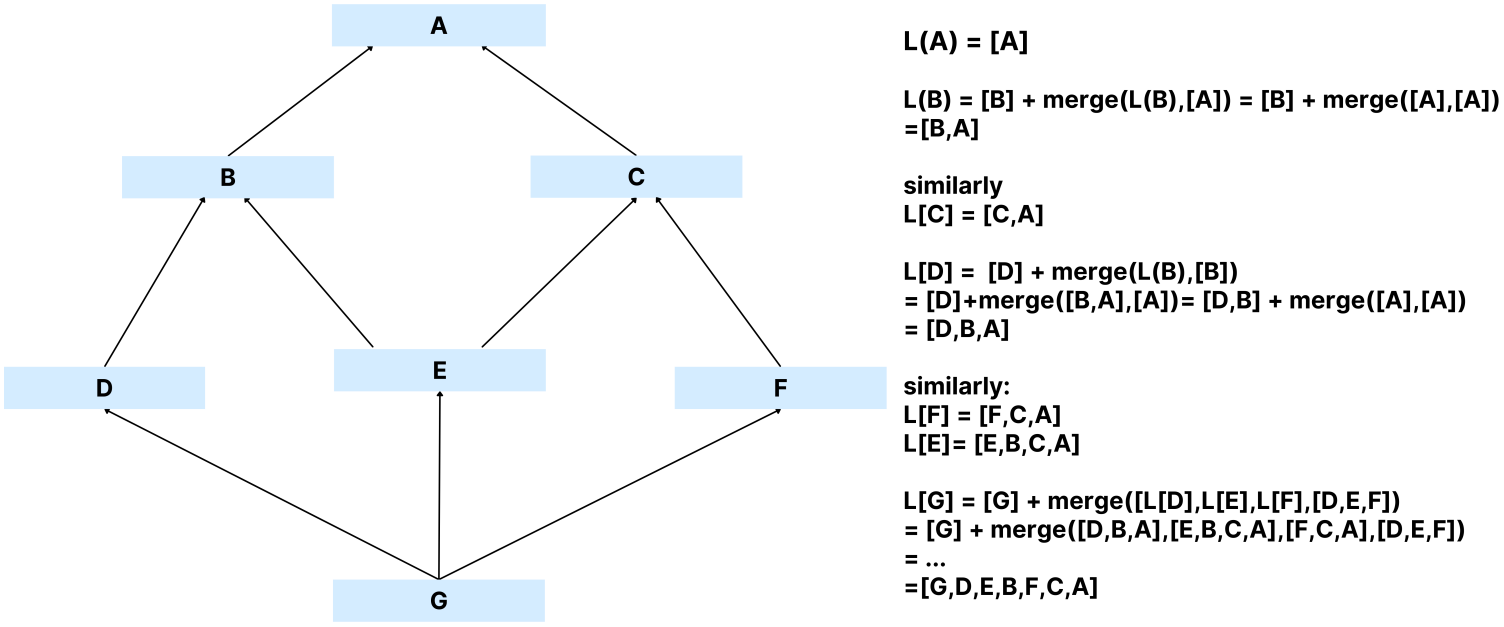

Figure 2.2: Example of C3 algorithm

This algorithm is a good example of a tool that can be used in a strategy to help manage both types of conflicts, especially sequential ones, as it can do the linearization and directly give the right feature activation order.

Topological Sort

We will explain the topological sort algorithm of Kahn's [13]. This will help us in the thesis to implement the algorithm behind one of our strategies. For this algorithm to work, it is necessary to have a Directed Acyclic Graph (DAG). The algorithm maintains a list of nodes that do not have any predecessors (we will call them orphans) and maintain for each node its number of remaining predecessors. In each iteration, the algorithm removes the next orphan in the list to add it to the topological order list. Then the orphan is removed with its edges from the graph, reducing by 1 the number of predecessors of its children. If the last step created new orphans (0 predecessors left), then they are added to the orphan list. The algorithm continues those steps until all nodes are explored and added in the topological order or if a loop is detected. The latter will raise an exception and indicate where the loop occurred. This algorithm of topological sort is simple to understand and can be used to resolve sequential conflicts as we will see in section 5.5.

Operator precedence in Prolog

Prolog (PROgramming in LOGic)[19] is one of the first logic programming languages and is now mainly used in artificial intelligence. It uses a declarative approach by giving the program relations and rules in the form of logical clauses. The computation is then done by using the defined relations and rules.

There exists multiple types of operators from basic arithmetic operators such as +, -, * to comparison operators >, <, =. Developers can define new operators which are explained in the book "Learn Prolog Now!" [4]. Just like arithmetic operators, Prolog operators must be operated in a certain order to avoid ambiguity. When doing 3 + 7 * 2, we are aware that we will first multiply 7 and 2 and then add 3. To be able to fix the order of operations in Prolog, each operator possesses a precedence score (between 0 and 1200), the higher the score, the greater the precedence. When resolving an expression, the operators with the highest precedence score will be operated first. The precedence is simply the priority order in which the operators are used. This mechanism is also important as Prolog allow developers to create new operators. The developers can define the precedence score they want depending on the newly created operators. When two operators have the same precedence, the behaviour is undefined as it depends on the Prolog implementation. However most of the time it is done from left to right order. We will explain how we took inspiration from this mechanism to design one of our strategies in section 4.4.

Aspect-oriented programming

One paradigm somewhat akin to FBCOP is that of Aspect-Oriented Programming (AOP) [17, 3]. It is a paradigm where developers can define "aspects" which are a kind of crosscutting modules that will inject (or "weave") code where it is needed into a program. It is done either during compilation or runtime and it allows to improve code maintainability and modularity. Aspects allow us to define crosscutting concerns in a modular way which means separating the code into more readable parts by separating the different concerns. When creating aspects, similar conflicts to the ones we will discover in adaptive systems can happen. Indeed, the order in which different aspects will inject code into the base system is essential, as different weaving orders can lead to different behaviour and therefore to sequential conflicts.

Possible solutions for this problem from various AOP frameworks will be discussed in the sequential conflict section. More precisely, we will talk about an extension of Java: AspectJ [16] and how it copes with the code injection order.

In AspectJ, every aspect has its own precedence which is similar to Prolog. The difference is that there is no numerical score for the precedence. Here, the precedence of an aspect is relative to other aspects. When multiple aspects inject code at the same location (join point), it is necessary to determine which piece of code is injected first. The injected piece of code is called advice in AspectJ.

As explained in the overview paper on AspectJ [16], for two pieces of advice a1 and a2 from two Aspects A1 and A2 that are injected in the same join point, multiple behaviours can occur:

- If the two pieces of advice come from the same Aspect (when A1 = A2), then the code will be injected in the same order that it is written inside that Aspect.
- If the two pieces of advice come from different Aspects, it will first look if one aspect extends the other. If it is the case then the refined aspect will take precedence as it is more specific.
- An aspect can also have the keyword "dominates". If A1 dominates A2, then A1 will take precedence and be executed before A2. This gives the developers a tool to define their own precedence order without extension between the aspects.
- Finally, in any other case, the precedence order is undefined which means that both aspects A1 and A2 are independent and can be injected in any order.

Issues with layer-based context-oriented languages

RubyCOP is implementing a specific COP approach which is Feature-based Context-Oriented Programming (FBCOP) that we already explained in Section 2.1. The original COP approach is using layer abstraction[11]. Layers group behavioural variations and can be activated or deactivated depending on the scope of activation assigned. However, this approach has some inconveniences.

For example, there is only one way to activate a layer but the interaction between layers can be intertwined and complex. To activate those layers, the developers need to find a way to define the activation layer without conflict. Another problem is the dependence between the main code and the layers. It is very similar to the aspect-oriented fragile pointcut problem [15] [25].

Solutions have been explored in the work of Cardozo and al [18]. They provided an Expressive and Modular layer Activation (EMA). It includes an API allowing developers to define more complex scoping for layers as well as interfaces between layers and the base code to avoid fragile pointcut. The layer interaction is kind of similar to our incompatibility conflicts and the solution in EMA could be used to partially solve them. We will explore this eventuality in the future work Section 7.

Reuse contracts

Reuse contracts have been introduced as means of facilitating software reuse. When developing new functionalities based on the already existing classes and methods of the main software, developers must be aware that this software is subject to changes. Changes can completely make the new extension obsolete and unusable by the end user. This is why Reuse contracts have been created. Introduced by Mens and al in their work on Reuse Contracts [24], Reuse contracts introduce a new set of operators and interface descriptions for managing change in software. Operators can be defined as follow: extension, refinement and concretisation. By adding these new operators and descriptions in the development of the main software and its extensions, it will indicate to the developer if any changes in the previous update are incompatible with his extension. The reuse contract is mainly used for conflicts with heritage property involving the parent classes. If the parent class change the type of return, delete or change the behaviour of a method, the children's classes could be affected by that change. We could use it as a detection tool for detecting, dynamically, when a feature enter in conflict with another one that is deployed after a change of context and thus incompatibility conflicts.

Chapter 3

Strategies for incompatibility conflicts

As we have seen in the introduction, an incompatibility conflict occurs when two contexts activated together, have incompatible features. For example, if one feature in a smart home system needs water in order to work (e.g. sprinklers to extinguish a fire) and another feature cuts the water supply (e.g. in case of a flood), we have an incompatibility conflict. This type of conflict is hard to detect without the right tools and often leads to unexpected application behaviour or even makes the program crash. To avoid such problems, we can use different strategies. A strategy dictates how the software system should behave in case of conflicts in order to avoid them. In this chapter, we first define different possible strategies, then we discuss how they perform in various case studies, and finally, we analyse some of the main strengths and weaknesses of each strategy.

3.1 Design Space

To frame and compare the different strategies we use the following design space which is an extended version of the one proposed by Mens et al [21] presented in Table 3.1

Figure 3.1: Design Space of conflicts

Axes

The resulting design space comprises 7 axes :

- How to resolve: This axis focuses on how we will resolve the conflict, this axis depends on the others; for example, if one conflict is caused by features and the conflict must be resolved dynamically, then the resulting strategy will be different.
- How to detect: This axis represents the detection tools. Those tools will not be able to resolve the conflict on their own but it is a great help for developers to detect where a conflict can occur. After using a detection tool, the developer can decide to resolve statically conflict by using, for example, a table or a graph.
- Who: This axis represents who should resolve the conflict. It could be either the end user, the developer/designer or the system. We can even imagine one possibility where a classifier decides what strategy should be used to resolve the conflict.
- When: This axis addresses when the conflict is going to be resolved, it is either statically, by knowing in advance how to resolve the conflict or dynamically, where the system will resolve the conflict in running time. We can also do both, a combination of the two.
- Where: This dimension focuses on where the conflict occurs. It could be caused by an error on the sensors, a Designer's/Developer's error or an error in the application. We could use different strategies depending on where the conflict happened.
- What: What causes the conflict in the program? A conflict can occur because of multiple components. It can be caused by the feature model, the context model, the mapping between the two, and even the constraints the developer puts on those components. Knowing what is the root of the conflict can help us to know which strategy will fit best.
- Why: The final axis tries to provide an answer to Why the conflict occurred. It has to take into account all the other axes in order to determine an accurate explanation. This axis can help the designer to analyse what specific resolution strategy would be most appropriate to resolve the conflict. Either the developer needs to analyse the conflict or we can imagine a self-adaptive system that uses self-explanation techniques. [2]

How to use the design space

This design space is used to classify all the possible strategies to solve conflicts or detect them. Defining a new strategy should answer the key questions asked by the axes, the resulting strategy will be named on the "How" axes.
For example, the prioritising strategy answers :

- Who : The developer/designer
- When : Dynamically
- Where : Application-Specific
- What : Features/Contexts
- Why : User analysis
- How to resolve : Prioritising

All the possible strategies should be able to fit inside this design space. It has to be used as a tool to define old or new strategies and remind developers what are the important sides of the strategies.

3.2 Detection of conflicts

Regarding the "How to detect" axis, there are essentially two ways to detect conflicts, either dynamically when the conflict occurs and there are no solutions defined for this conflict, or by using static detection tools to detect conflicts before they happen. Most of these detection tools are used statically but some of them can be run dynamically.

Developers should use static detection tools to detect and remove conflicts before they happen. But they should also use dynamic detection tools paired with conflict-resolution strategies to make the strategies more efficient. For example, we can use a PetriNet to detect conflicts and pair it with a rollback strategy so that the system can go back to a previous stable state when detecting a conflict, as shown by Cardozo et al [6]. We also explore the eventuality of using Reuse contract [24] in the section 2.2. Reuse contracts could be used to dynamically detect incompatibility conflict by introducing new keywords when defining new features. When a new feature is deployed, we will be noticed if it interacts with other features and this interaction cause a conflict. We could then apply our strategy based on this feedback. It is important to remember that these tools cannot resolve conflicts by themselves and need strategies to resolve them.

There are different types of detection approaches, which can be formal methods (e.g. based on PetriNets or SAT solving), models, rule-based and transition systems [5]. Because it is not the main goal of this thesis, we will not explain them in detail here, as the main focus of this chapter is the resolution of incompatibility conflicts using strategies and not detecting them. There are no incompatibility conflicts detection tools implemented in RubyCOP yet, and the implementation of such tools could be a thesis in itself, so we will not discuss it further.

3.3 Listing of strategies

We define here all the known strategies to resolve conflicts in feature-based context-oriented software. Most of those strategies were already defined by Mens et al. [21].

Using runtime exceptions

When a conflict happens, the system only throws an exception at runtime. There can be two types of runtime exceptions. Either the developer was not aware of the conflicts, in which case the system just crashes and throws an error. Or the developer is aware of the conflicts and creates user-defined errors. Only in this case, we can call it a strategy as the exceptions thrown could be handled by other parts of the system.

Prompt Strategy

The prompt Strategy will try to adopt new features as soon as their contexts are activated. This strategy basically prioritises every time the features of the most recently activated context so that the running program becomes more coherent to the real-world situation.

Loyal Strategy

The Loyal Strategy consists of adopting new features only when the currently executing context exits. As opposed to the Prompt Strategy, thus prioritising the features of the already active contexts. All the new incoming activation will simply be ignored as conflicts occur.

Prompt/Loyal Strategy

The prompt/loyal strategy is a mix of both strategies. It will try to adopt a new context as soon as the currently executing feature is done. It is done by postponing the new context and waiting before activating it by using a simple queue of contexts. The system will remain loyal to the first activated feature and will still promptly activate the new incoming features in the order of arrival.

Revert to default

As soon as a conflict occurs, the system will revert to its default behaviour; For example, a more generic version where the conflicting features are removed. For this strategy to work, the system developer must think of and implement a correct default behaviour.

Rollback

When a conflict occurs, revert to the previous state that was stable. Multiple types of rollback are possible. We could, for example, only rollback the conflicting feature, that we tried to activate, but we could also do a rollback where we will revert the features deployed and the contexts activated to a stable known state. You can see this as a complete rollback. Another way to do it could also be using the rollback only on the conflicting contexts and getting back to a stable state before even deploying the new corresponding features.

Prioritising

By using a prioritising strategy, we can sort the different features by significance and thus, avoid conflict. For example, a high-priority feature is executed when a conflicting feature is deployed with a lower priority. The newly deployed feature will just be ignored since it has a lower priority, on the other hand, if the new feature had a higher priority, it will overwrite the executing one, taking its place. Prioritising could be defined by the developer using, for example, an explicit annotation. It could also be imposed by the language using an implicit linearisation strategy or by defining an activation policy that will define which context or feature the program should activate first.

User-driven

When a conflict arises, the system could also let the user choose which feature fits best. This is done by requesting the users what behaviour the application should

follow or by collecting information on them. This strategy is based on the fact that users should know what is best for themselves and therefore allows a service tailored to their needs.

User as a Context

This strategy is conceptually the same as the previous one, but the modelisation is different. This strategy uses context-adaptive mechanisms to consider the users and their preferences as contexts. Those contexts are different from one user to another and give a personalized system depending on the needs of the users. It can also get the user's choice either by requesting it or by collecting it automatically. Because both the "user-driven" and "user as a context" strategies are similar and produce the same output for the users: When comparing the strategies, we will distinguish them by considering that the user-driven strategy requests the user's preference whenever a conflict occurs. And that User as a context automatically collects the users' information/preference if they allowed it.

Developer-driven

The purpose of this strategy is to let the developer define which strategies should be used to solve conflicts. The developer has to know beforehand which conflicts can occur in order to choose the best strategies to solve them. It can be done by using detection tools, as explained in section 1.2. After running those tools, he can specifically address the different conflicts normally, in a table or a graph-based representation of the feature/context that could enter into conflict with each other and explicitly define which strategies are going to be used. The developer can use different strategies depending on the type of conflict, thus making the Developer-driven a meta-strategy.

3.4 Case studies

In order to analyse the strategies previously listed, we define 4 case studies to cover a wide range of possible cases. The cases are examples of conflicts happening in software using the feature-based context-oriented paradigm.

Fire versus Flood

Assume a home automation system able to detect fires and water leaks and that can adapt its behaviour to address the emergency; Duhoux presented a similar case in his thesis [9]. In the case of a fire, the system must trigger the sprinklers

and send an alarm to the fire brigade. In the case of a water leak, it must turn off the main water supply. When only one such event occurs, the system is aware of which action it must undertake. But when both the fire and the water leaks are detected, a conflict occurs because the sprinklers cannot be activated if the main water supply is turned off. We consider for this example that both contexts can occur in any order so the strategy used to resolve it should work independently of the order of occurrence.

Day and Night

Assume a smart messaging system [9] that can detect whether it is the day or the night and activates the feature light mode or dark mode respectively. Light mode will change the colour of the boxes to white and the text to black, whereas dark mode will do the opposite. Those two features are incompatible together, they should not be activated simultaneously. But also it does not make sense to have both Day and Night contexts activated at the same time, so both contexts are defined as incompatible in the Context model.
In this case of study, we will consider that one context (either day or night), does not deactivate when the other activates, thus creating a conflict. This could happen when we use two different sensors to detect the same thing, for example using two different clocks to detect the day and night or if one of the sensors is defective.

Important call in Do Not Disturb mode (DND)

Assume a smart messaging system [9] with two features: Important call and do not disturb mode. The first one allows giving more importance to a call and the other one blocks the notification sounds and ringtones when the user is busy and should not be disturbed. A conflict happens when someone makes an important call to someone who is in do not disturb mode because depending on the situation, some people may want to be notified of important calls even in the do not disturb mode while others do not.

3.5 Results Analysis

In this section, we analyse the outcome of the conflict resolution for each case depending on the strategy used.

Using runtime exceptions

- Fire/Flood : The house will be flooded and burned down because the system will throw an Exception like "UncompatibleFeaturesException", when both the flood and the fire contexts are activated, as we cannot activate the sprinklers while the water is deactivated.
- Day/Night : The smart messenger will throw an Exception like "UncompatibleContextsException" when both contexts are activated. This will make the smart messenger crash or have faulty behaviour unless the exception was handled.
- Important/DND : The smart messenger will throw an Exception like "UncompatibleFeaturesException" when the important call is made while in do not disturb mode. This will make the smart messenger crash or have faulty behaviour unless the exception was handled.

Prompt Strategy

- Fire/Flood : If we apply this strategy to the first case, either the house will burn or be flooded. If the fire is detected first, the house system will turn on the sprinklers, but when the flood occurs, our prompt strategy will adopt the change as soon as the flood is detected, cutting off the water and stopping the fire extinguishing. If the opposite happens, first the flood and then the fire, the house will be flooded.
- Day/Night : For the day and night case, the prompt strategy will accept the new changing context as soon as it appears. For example, if the context day does not deactivate and the night context activates because of a faulty sensor, the prompt strategy will activate the night context's features.
- Important/DND : For the third case: important call, the prompt strategy will ignore the do not disturb mode and the phone will ring.

Loyal Strategy

- Fire/Flood : In our first case, if the fire occurs first, the fire will be extinguished, the flood will be ignored, and vice versa.
- Day/Night : For the second case, the program will remain loyal to the first context sensed and if this one never exits, the night or day mode will never be activated.

- Important/DND : The loyal strategy will remain loyal to the do not disturb mode and will ignore the important call.

Prompt/Loyal Strategy

- Fire/Flood : If the fire occurs first and then the flood, the flood-related features will be pending until the fire is extinguished then the flood will be stopped. A good point of this strategy is that it will solve both the fire and the flood, as opposed to the prompt and the loyal strategies.
- Day/Night : The loyal/prompt strategy will not be efficient. For example, if the day remains activated permanently, the night context will be postponed forever and will never be activated since the day context never exits.
- Important/DND : The prompt/loyal strategy will remain loyal to the "do not disturb mode" and the important call will be postponed until the user disables the mode.

Revert to default

- Fire/Flood : The house will be flooded and burned down. Except if the default behaviour is calling emergencies, in this case, the house will be flooded and burned down only if the emergency services do not arrive in time.
- Day/Night : The system will go to default mode and be usable, but the colour scheme will not always match the day or night mode. The default mode will probably be either the day or night mode.
- Important/DND: This is the worst case as the smart messenger will go back to default when the conflict happens, it means that the user will not get the important call, and even worse, he will not be in do not disturb mode anymore. Unless the important call or the do not disturb are activated in the default mode.

Rollback Strategy

- Fire/Flood : When the conflict happens, the system will go back to the previous state without conflict. The house will then suffer only one of the two disasters, either being burned down or flooded depending on which context is activated first.

- Day/Night : The smart messenger will always stay either in light mode or dark mode because it will go back to the previous state every time a conflict happens and a conflict will happen every time the system tries to activate the opposite context.
- Important/DND : The rollback strategy will choose to deactivate the important call and keep the do not disturb mode. This means that the smart messenger will not receive the important call and will stay in "do not disturb" mode.

Prioritising Strategy

- Fire/Flood : We can prioritise the features related to the fire context, for example, start the sprinklers and call the fire force. The flood context's feature will have a lower priority. In case of conflict, the fire will be solved first, and then the flood.
- Day/Night : The problem here is that if one context with a high-priority feature stays active permanently, we will be in a livelock situation and the night or the day context will never be able to activate its features
- Important/DND : The result will depend on the priority we put in the important call feature. If it has a higher priority than the feature of do not disturb context, the user will receive the phone call even in the do not disturb mode. If it is not, the phone call will be ignored.

User-driven (requesting user's preference)

- Fire/Flood : The user can choose what is best between having the house burned down or flooded. So the outcome will be the best as the user chooses what is better, for example depending on what insurance policy he has. In this case, the user should normally choose to extinguish the fire.
- Day/Night : The user can choose which mode he prefers between light and dark mode whenever the conflict happens. This means that the user can always choose the correct mode depending on the time of the day.
- Important/DND : The user can choose if he wants to be disturbed or not during the meeting so the outcome will fit the user's needs. It may not work the first time as the user has to see the request during his meeting but only after it, but the smart messenger will remember his choice and will know for the next time this conflict happens.

User as a Context (Collecting user's preference)

- Fire/Flood : By asking about the preferences beforehand, the system can know in advance if the user would prefer having his house burned down or flooded, so when the conflict happens, the system will know which adaptation to choose. The designer of the smart home should know in advance which conflicts can happen so he can ask the user's preference related to those conflicts.

- Day/Night : The system knows which mode the user prefers so it will only activate either day or night and deactivate the other one depending on the user's preferences.

- Important/DND : The user can tell the system if he wants to be disturbed by important calls while in do not disturb mode. The outcome will be the one chosen by the user. The system could also learn which outcome is the best by analysing the behaviour of the user. If for example the user always responds to important calls even in do not disturb mode then those important calls will always notify the user. On the other hand, if the user never answers the phone in do not disturb mode then the important calls will be muted.

Developer-driven

The developer can choose the best strategies for each of the conflicts, either in advance or when the conflicts are detected dynamically. He will for example choose Prompt-Loyal for the first case, Prompt for the second case and Prioritising for the third case. Because it is a meta-strategy, we did not put this strategy in the comparison table as it cannot really be compared with non-meta strategies.

3.6 Summary of strategies

By using the previous results, we created a table that summarises the outcome of each case study for each strategy used. We used +, =, and - to represent respectively best case, good case, middle case, bad case, and worst case. It is shown in Table 3.1

Strategy/Study Cases	Flood/Fire	Day or Night	Emergency/DND
Using runtime exceptions	-	-	-
Prompt Strategy	=	+	=
Loyal Strategy	=	-	=
Prompt/Loyal Strategy	+	-	=
Revert to delfaut	-	-	-
Rollback Strategy	=	-	=
Prioritizing Strategy	+	-	+
User-driven	+	+	+
User as a Context	+	+	+

Table 3.1: Ranking strategies table

Flood/Fire

- + : One or both of the disasters are solved and the outcome is predictable, which means that the designer or the user could choose which outcome is best.
- = : Only one of the disasters is prevented but the outcome is unpredictable.
- - : The house is flooded and burned down.
- – : The smart messenger crashes or throws a runtime exception.

Day/Night

- + : There is no livelock situation, therefore the program runs as expected. The day and night contexts are activated at the right time.
- = : The same mode is always active, but it is the user's favourite one.
- - : The same mode is always active but unpredictable.
- – : The smart messenger crashes or throws a runtime exception.

Important/DND

- + : The outcome is chosen either by the user or the designer.
- = : The outcome is unpredictable, either the phone rings or does not.
- - : The smart messenger exits the "do not disturb" mode.
- – : The smart messenger crashes or throws a runtime exception.

3.7 Discussion about strategies

In this section, we step back and analyse each strategy in a more general way. We answer the question "Is this strategy always working?" and we explain why.

Using runtime exceptions

This strategy is only useful for developers to debug their Feature-Based Context-Oriented programs. Apart from that, this strategy does not bring anything else than returning an exception and should therefore never be used in a final program as it only stops your program when an exception occurs.

Prompt Strategy

The prompt strategy is a kind of a double edge sword, it works well in many case studies including ours but we can imagine a different case where an important feature/context enters in conflict with a non-crucial feature/context. For example, a fire occurs and a water restriction feature activates and limits the amount of water used. The two will enter into conflict and if the water restriction feature is activated lastly, the fire could not be extinguished correctly. But, it can be a good strategy for equally important features/contexts and easily solves conflicts and it respects the context-oriented paradigms as it adapts to the newest contexts.

Loyal Strategy

The Loyal strategy has the same defect as the Prompt one, with both a crucial and a non-crucial features/contexts conflict, it can lead to non-optimal behaviour as we have seen in the prompt strategy discussion; If the water restriction is activated first, and then a fire occurs, the strategy will remain loyal to the first feature at the risk of burning the whole house. Another big problem of the Loyal strategy is that it can lead to livelock situations. If a feature/context never deactivates, the loyal strategy will remain stuck without including the other adaptations of the program.

Prompt/Loyal Strategy

The Prompt/Loyal strategy is a combination of both strategies, and thus, it inherits its flaws. It can suffer from livelock since we remain loyal to the first feature/context activated. By using a queue, it keeps in memory the future features/contexts that will be activated whereas, in case of a pure prompt or loyal strategy, we ignore incoming changes or forget the current feature. The Prompt/Loyal strategy is one of the few strategies that could resolve the two conflicts together, in the case

of the fire and the flood, since we use a queue, as soon as one feature is done, start the next one. It does not solve the live lock case and we cannot predict which conflict should be solved first. One of the solutions is to mix this strategy with prioritisation to solve the ordering and a timeout system to solve the livelock problem, we should not forget that using a timeout can lead to unsolved conflict: For the first case study, the fire could take too much time to be extinguished, the timeout occurs leading to ignore the fire and deploy the flood feature. This strategy is an interesting one but it could be hard to implement it because of those.

Revert to default

This strategy has a lot of drawbacks. First, in order to work, this strategy needs a default behaviour to be defined by the designer. The designer, therefore, needs a good understanding of the system and has to choose and implement a default behaviour. If this default behaviour is not well thought, it can lead to undesirable behaviour. Even if it is well built, it will not be perfect every time. A second drawback is that it goes a bit against the Context-oriented paradigm where the system should adapt correctly to the most appropriate contexts since in case of conflicts it will fall back to the default case that is less specific.
Here, whenever a conflict happens, the strategy makes the system go back to the default behaviour. Because of that, the system does not adapt every time to the new contexts activation and deactivation.
Because of these reasons, this strategy should not be used or only as a last resort in combination with other strategies.

Rollback

The rollback strategy is similar to the revert to default. It has the same problem: it does not adapt to new contexts as it just goes back to a previous state when a conflict happens. Another problem is that the developer needs to implement states in order to use this strategy. This strategy is better than the return to default as it is adapting better regards to the contexts but still, when conflicts occur, it will just go back to a stable state which means that in most cases, it will not fit the user's needs. Note that after rolling back the system's state will be inconsistent with the real world around it. Because of these reasons, this strategy should not be used or only as a last resort in combination with other strategies

Prioritising

The prioritising strategy seems interesting when seeing the table. Indeed, this strategy allows the developer to give more importance to certain adaptations than

others. This strategy is therefore interesting in situations where some adaptations are significantly more important than others. This strategy will be less effective if the outcome of the adaptations is subjective and if the best adaptation depends on one user to another. This strategy also needs a tie-breaker in the case where two adaptations have the same priority. The developer should use another strategy to resolve the conflicts inside the prioritising strategy such as prompt strategy which means prioritising the newest context first.
We think this strategy is essential, as it gives the developer the freedom to choose which adaptations are more important. This strategy also looks promising as it is important to be able to give priority when some adaptations are critical such as the house on fire.

User-driven

This strategy looks perfect when looking at the comparison table. It is because the table only looks at the outcome of the conflict resolution. And because we use feature-based context-oriented programming, we want the system's behaviour to fit each user individually. But there are huge drawbacks. First, this strategy is extremely intrusive for the user as it requires explicit actions from him. It also requires the user to be ready to give his preferences whenever a conflict occurs, which means that if he does not use his phone while his house is burning and flooding, he could miss the alert and the conflict would not be resolved. Even if the strategy uses user preferences instead of explicit requests, it still requires the user to give a lot of personal information. Because of that, this strategy is not as good as it seems and should not be used unless it really helps to know the user's preferences in case of subjective outcomes related to the adaptations. This strategy is also useful for extreme conflicts that do not happen often as the user will not be disturbed by a lot of unimportant choices. (in the case of requests sent to the user)

User as a Context

This strategy has the same drawbacks as the previous one. The idea of using the user as a context is a nice idea, especially in our FBCOP paradigm. But in reality, it is hard to make it happen. Just like the previous one, it is too intrusive or collects too much data in today's world where privacy is so important. This strategy should not be used unless the data collected about the user does not have to be private or if the user agrees in letting the system record a lot of personal data.

Developer-driven

The Developer's Choice strategy is one of the most interesting but also one of the most arduous to set up. The developer must first list all the possible conflicts in the program using a static detector, and then choose the optimal strategy for these conflicts. Note that using different strategies can also bring conflicts between strategies. We can imagine for example that for one conflict we use a Prompt/Loyal strategy and, while executing, another conflict occurs where we solve it by using the revert to default strategy. While we are reverting to a default behaviour, we will interrupt and lose all the adaptation in memory for the Prompt/Loyal. The developer has to choose meticulously and avoid those conflicts. If this strategy is correctly used, it could be one of the most optimal strategies, but it comes with a high intellectual cost and may be very hard to test if it works well with all the possible combinations.

3.8 Conclusion

During this chapter, we defined the incompatibility conflicts, the Design Space and the strategies that can be used to avoid those conflicts; we analysed them to decide which strategies seem the most interesting to implement. After analysing them with the case studies, we concluded that the most interesting strategies to study are prioritising with prompt as a tie-breaker, Loyal-Prompt, Rollback and user-driven/user as a context. But also Developer's Choice strategy which requires other strategies to be already implemented first.

Because there are no detection tools implemented in RubyCOP, it makes it difficult to implement strategies for incompatibility conflicts that could manage all conflicts. The system cannot know when such conflicts occur. Creating such detection tools will cost a considerable amount of time. For this reason, we will focus only on sequential conflicts for the rest of the thesis.

Chapter 4

Sequential conflicts

In the last chapter, we defined the notion of incompatibility conflict, when such conflicts occur and what kind of strategies would be able to manage such conflicts. In this chapter, we will take a look at sequential conflicts. Those conflicts are related to the feature activation order. Such conflicts occur when for two executions with the same input, but different activation orders, the behaviour of the adaptive system is not the same.

The feature activation order can depend on the context activation order in some cases. However, all strategies presented in this chapter will focus only on the feature activation order by making it independent of the context activation order. This will make the problem easier if we only look at features and not at both features and contexts.

We will also introduce the case study we chose to explore this conflict in detail. Finally, we will explain the strategies that will help us manage sequential conflicts by adjusting the order of activation of the different features, and how they are implemented.

In the following chapters, we will use the word **precedence**. Having precedence over a feature is defined as being prioritised inside the activation order over another feature. We define precedence between two features f1 and f2 as follows:
If f1 has precedence over f2 then f1 will always be activated before f2, regardless of the configuration of the feature model. It is only a time-related priority and not the feature itself that is prioritised. f1 is not more important than f2, it is only activated before.

4.1 Running example

This chapter will use a running example to illustrate the implemented strategies. The case study is a variant of the smart messaging system, inspired by the running example described in Benoît Duhoux's PhD thesis [9]. This smart messaging system allows users to create chats, and use them to send and receive messages to/from other users. This messaging system is adaptive: depending on different contexts, it can adapt to the activity, device and age of the user and features vary from changing the sound of notifications to the type of messages sent.

Messages can contain different types of information, either a text, one or more files, a location or a combination of them. By default, only texts and documents can be sent, but depending on the context, you can add more data to messages. The features **location** can be activated if the phone is equipped with a Gps, **pictures** and **videos** with a camera, **vocal messages** with a microphone and messages will be **compressed** if the network is of bad quality.
The system adapts to the type of device, a phone will have a different layout, more simplistic, than a tablet or a computer.
The system can also adapt to the time of day, with light and dark modes that will adapt in order to be less exhausting for the eyes of the user.
Another main feature is the users' status, which is defined depending on the current activity they are doing. He will be Connected when using the app, Busy when driving and Do not disturb when in a meeting. The status will change the notification sounds to Standard, Mute or Audio reading correspondingly.
Finally, depending if a child or an adult uses the application, the permissions will not be the same. Children will not be able to share their location for security purposes. And also, parental control can be set up so that parents can monitor their child's activity.
This running example will be fully explained in chapter 6 where it will become a case study for the validation process.

4.2 Example of sequential conflict

In this section, we will present a concrete situation where sequential conflicts can occur. We will use a simplified and modified version of our running example, the smart messaging system, by only showing the contexts and features that play a role in the conflict.

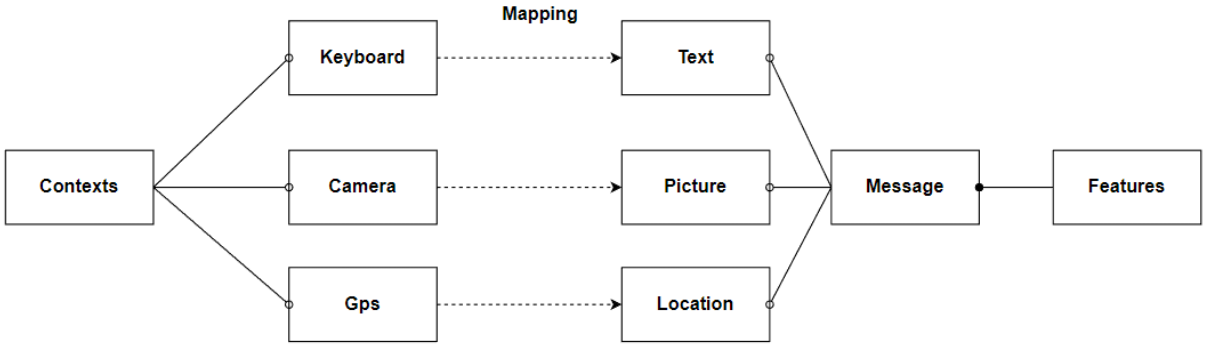

Figure 4.1: Simple example of context-feature model

When all contexts, **keyboard**, **Camera** and **Gps** are activated without strategies, the related features will be activated in the same order as their corresponding contexts by following the mapping. The configuration "Keyboard-Gps-Camera-Text-Picture-Location" (all activated features and contexts), can give different outcomes, depending on which context was activated first. This is a real problem for the users as their messenger's UI will change depending on the context, which will be confusing or annoying. The UI should be always the same and independent of the context activation order.

As we can see in Figure 4.2, even with the same configuration, the behaviour of the smart messaging system can be totally different. One time the text is above the picture and the location but sometimes it could be the location first or any combination of Text-Picture-Location depending on the activation order of the contexts. The three examples are all valid and have the same configuration, making a perfect example of sequential conflicts. Indeed, for the same configuration, we have more than one execution leading to a different behaviour, therefore we have a sequential conflict.

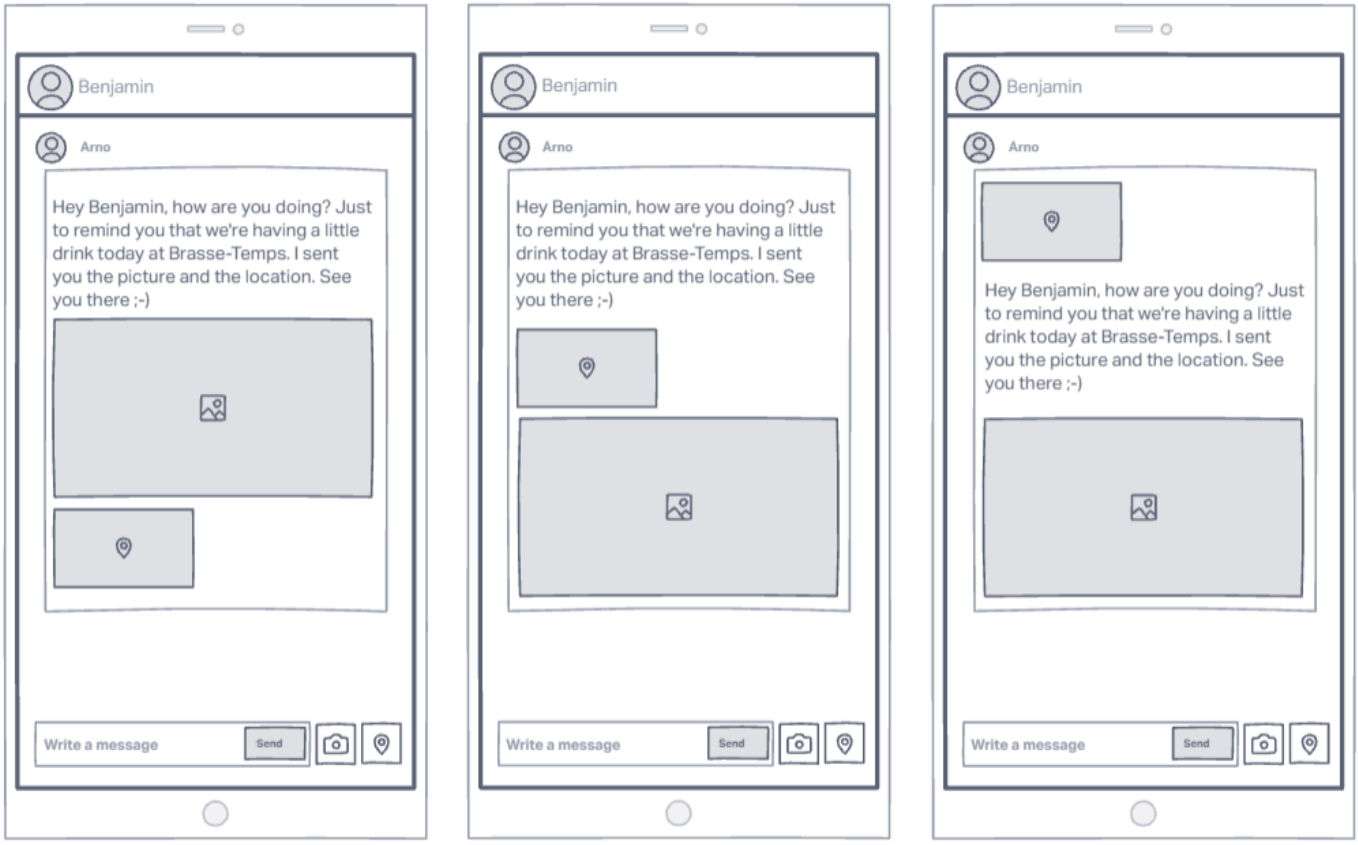

Figure 4.2: Example of sequential conflict

4.3 Default behaviour of RubyCOP

The *proceed* mechanism is crucial in RubyCOP, Duhoux [9] defines this mechanism as follows: "In a feature, when a method is using a proceed keyword, it will install the properties of its previous version from the last feature deployed". This mechanism builds the features incrementally by always taking the method of the latest installed feature. Building incrementally makes the feature activation order important because it can change the behaviour of the application, especially with visual features such as the layout.
By default in RubyCOP, the mandatory contexts are activated in the order the developers wrote them which corresponds to the DFS order. Every time a context is activated, it will activate its corresponding features in the mapping order. For example, if the mapping is defined as: "Camera -> Picture, Video" then Picture will be activated before Video. Controlling the activation order with the mapping can be useful, but it is not sufficient to resolve sequential conflicts as we have

seen in the previous section 4.2. Features are built incrementally with the *proceed* mechanism. Therefore their activation order changes the behaviour of the application. But, because the contexts activate the features, the order of activation of the contexts controls the feature activation order. This is a problem in adaptive systems as contexts can be activated or deactivated at any time in any order. The same configuration can therefore lead to different behaviour depending on how the contexts were activated.

To resolve such sequential conflicts, developers will use strategies. By doing so, the behaviour of an application will always stay the same for the same configuration. Some strategies even give developers the flexibility to choose their preferred activation order.

4.4 Strategies

Strategies for resolving sequential conflicts differ from the ones used for incompatibility conflicts. While for incompatibility conflicts we delay or ignore the activation of contexts, for sequential conflict strategies, we arrange a new order of feature activation. This activation order will be defined by using strategies. Strategies give developers of the adaptive system control over what order they prefer and thus, how the application will behave.

We decided to create strategies that focus on changing the feature activation order while making it independent of the order of contexts to simplify the problem.

Numerical precedence

For this ordering strategy, developers can define numerical values for the features they implemented. By doing so, the system will activate them following the ascending order of those values, activating features with the lowest values first and the highest value last.

The constraint with this approach is that the parent feature must always be activated before their children in order to have a valid execution. Otherwise, a child could be activated before its parent which may cause a failure if the child has a super call. Not only a super call can make the application fail but also the *proceed* mechanism. Indeed, using the *proceed* mechanism without guaranteeing the hierarchical order can lead to crashes or faulty behaviours. For example, when a method proceeds but there are no previously installed features with this method, the system will throw an error. Another example would be visually the child's

layout would be above its parent. This hierarchical property is not only necessary for this strategy but also for all the other strategies for the same reasons.

This strategy also needs a tie-breaker when the developer gives the same precedence score to multiple features. For example, the DFS order, which we will talk about later in Section 4.4, can serve as a tie-breaker.

This strategy is similar to how operator precedence works in Prolog, by giving numerical precedence scores. [4]

Relative precedence

This strategy allows the developer to define relations between features to arrange their activation order. When talking about **relations**, we are talking about precedence relations between two features f1 and f2 where f1 takes precedence over f2. This means that when this relation is defined by the developer, f1 will always be activated before f2. In this thesis, the precedence order will be represented with the ">" symbol where f1 > f2 means that f1 takes precedence over f2.

For example, with the relation $\mathbf{A > B}$, feature A will be activated before feature B. Note that we have to be careful because the same constraint as the numerical precedence strategy applies here: the parent features must be activated before its children. In the case: $\mathbf{B > A}$ relation, if A is the father of B, then this clause is not valid and will not be accepted, as it can make the system crash. The relations are also transitive, if $\mathbf{A > B}$ and $\mathbf{B > C}$, then $\mathbf{A > C}$.

This strategy is similar to precedence to the AspectJ extension where there is a keyword "dominates" to define precedence between two aspects [16].

Depth First Search Order

This strategy will use the feature model to arrange the feature activation order. It will follow the DFS order, which means, it will fully explore all paths in the feature model one by one. This ensures that we do not activate a child feature before its parent and since the order of activation is always fixed, we cannot have any different executions, leading us to no sequential conflict. This DFS order explore the model from top to bottom by going left first and then to the right. This way, the first defined features within the same depth (= the ones to the left) will be activated first. It can be used as a strategy on its own or as a tie-breaker for some strategies such as the numerical precedence strategy.

It is important to note that here the DFS order can be replaced by a BFS order and that will also resolve sequential conflicts. In this thesis, we use the DFS order instead of the BFS for several reasons that are explained in Section 5.3.

4.5 Conclusion

In this chapter, we explained the sequential conflicts, presented our running example and we gave concrete examples of such conflicts. We then explained theoretically what strategies can be used and how they would resolve the conflicts.
The next chapter will be about implementing those strategies in RubyCOP.

Chapter 5

Implementation

In this section, we will first describe the refresh mechanism, which is a mandatory procedure to change the feature activation order. Then we will explain the implementation of 3 sequential conflict resolution strategies. We decided to implement all the strategies described in the last section. We have also decided to change RubyCOP by using the DFS strategy by default so that there are no longer sequential conflicts that can happen.

5.1 Refresh mechanism

The order of activation in the latest version of RubyCOP is fixed: after a context is activated, the corresponding mapped features are activated and those stay active until the deactivation of the context.
But what happens if you want to activate a feature before another already activated feature? In such an environment, it is impossible to activate a new feature before another already activated feature. In other words, we cannot change the activation order of features that are currently executing. That is why we need a refresh mechanism. Its goal is to deactivate all activated features and reactivate them in the order chosen by one of the strategies that will include the newly activated features. This way, we can keep a fixed feature activation order no matter the combination of contexts. The refresh mechanism allows us to fix the order of activation with strategies and therefore, remove all sequential conflicts as long as the strategy used has a deterministic behaviour (always the same behaviour, no randomness).

When refreshing, the system deactivates all activated features and reactivates them within the same call. It is obviously mandatory to first deactivate and then reactivate. The order of reactivation is given by the strategy, for example, the numerical strategy will sort them by the precedence score. Also, the developer

should not forget when defining the precedence, that if a feature is activated before its parent or one of its ancestors, the program will likely crash or produce the wrong behaviour. This can happen because if the feature calls super() (proceed() in RubyCOP), its parent will not be defined, resulting in an exception.

In order to guarantee that each feature is activated after its parent when refreshing, the strategy used must reactivate them in the hierarchical order (parent features first then children features). All strategies that we implemented respect the hierarchical order or throw an exception if it is not the case. It is also important when deactivating features (before reactivating them), to deactivate them by doing the children first then the parents, in reversed hierarchical order. It is because some features call their parent features in their epilogue (a piece of code that runs when deactivating the feature). This requires the parent to be activated when deactivating the child feature.

In order to resolve sequential conflicts, strategies must use the refresh mechanism. This can only be done by respecting the hierarchical order otherwise the system could crash or have faulty behaviours.

It is important to note that only the activation order can lead to conflicts and needs a refresh in order to fix them. Whereas deactivating features does not require any refresh as they do not create conflicts. A refresh only occurs upon activation of new features. The only requirement when deactivating features, because of a context change, is that the feature deactivation should occur after the refresh. Otherwise, the refresh will not work because the system will try to refresh a deactivated feature.

5.2 Abstract Strategy

In order to implement the different strategies, we decided to implement an interface named Strategy that follows the Strategy design pattern [10]. By using an interface, it is possible to change the feature prioritisation strategy just by changing one line of code and it will automatically switch to a different strategy. It also allows the implementation of new strategies quickly without having to change the entire architecture of the code. To create a new strategy, we only need to make a new class that implements the interface Strategy. This interface only possesses one method called apply(aoe). It takes a list of AOE (action on entities, such as "activate - given feature"), and will sort this list in the order in which the AOEs should be executed. This sorting depends on the strategy used, as each of them implements a different method apply(aoe).

5.3 DFS order

The implementation of the depth-first-search is pretty straightforward and does not require any effort from the developer to resolve conflicts. After doing the refresh, this strategy will reactivate the features and activate the new features in DFS order, guaranteeing both respected hierarchical order and no sequential conflicts.

We choose to use this strategy by default in RubyCOP because it removes the sequential conflict and does not require the developer to do anything. The developer can change this default strategy to another one if he wants to customise the feature activation order. With this strategy by default, features will be activated following the DFS order: taking the first feature defined then its first child defined and so on. This ensures that hierarchical order is respected, no matter the shape of the feature model.
Listing 5.1 is a simplified version of our running example. As a reminder, when defining a feature, the arguments go as follows: type (context/feature, abstract or not), the reference of the feature, name of the feature and classes that the feature adapts or refines (only applicable for concrete features). The activation order when applying the DFS strategy will be (Root) > (FileType) > Document > Picture > Vocal > Video, as DFS visits each path of the model from left to right. Note that Root and FileType are between parenthesis in the activation order because they are abstract features.

```
class AppFeatureModelDeclaration < FeatureModelDeclaration
    include Singleton
    def initialize()
        super()
        _define_file_types()
        @root_feature.relation :Mandatory, [@file_type]
    end

    def _define_file_types()
        abstract_feature :@file_type, 'FileType'
        feature :@document, 'Document', [:MessageModel]
        feature :@picture, 'Picture', [:MessageModel]
        feature :@vocal, 'Vocal', [:MessageModel]
        feature :@video, 'Video', [:MessageModel]
        @file_type.relation :Mandatory, [@document]
        @file_type.relation :Optional, [@picture, @vocal, @video]
    end
end
```

Listing 5.1: app feature model example

DFS vs BFS

During this entire thesis, we only talk about a DFS implementation. It is important to note that it is entirely replaceable by a BFS implementation. The reason behind that is simple, DFS and BFS are search algorithms, where the goal is to travel the tree/model to find the features to activate. However, in our case, we want to explore every node of the whole model. No matter which algorithm is used, the same features will be explored and activated with the same time complexity but in a different activation order. There is another difference between both algorithms and this is the memory usage.

In terms of memory usage, DFS uses less memory than BFS. The reason is simple: for each new node, BFS has to keep in memory in a queue all the children nodes that it will have to explore afterwards. Whereas the DFS only keeps in a stack the current features and will explore all the paths across the model one by one. This memory difference is not noticeable at all in small models but should be considered nonetheless when building bigger models.

The other difference between both algorithms is the activation order. It is the most important property regarding sequential conflicts.
The BFS activates all the features layer by layer first so that the lower-depth features will be activated before the deepest ones.
On the other hand, the DFS activates the most left features first (the most left feature is the first declared in the model). By using the DFS order, we activate the parent and its successors one by one.
Since each child feature is a refinement of its parent feature, DFS activates features and their refinement consecutively.
To give an example, using our smart messaging system, we would activate groups of features together when using DFS such as: Notifications and all its successors' features, then Send, then Contact... Whereas with a BFS we would activate Notification, Send, Contact ... and then only their children layer by layer.
Contrary to BFS, DFS is not dependent on depths or layers. It makes it more convenient because it can change the activation order of non-sibling features only by changing the declaration order. There is no need to change the structure of the model.

For these reasons, we think that DFS is a better tie-breaker for our strategies compared to BFS.

5.4 Numerical precedence

This strategy allows the developer to put a numerical value (the precedence score) next to each feature he declared (The precedence score is 0 by default). After deactivating all the activated features in the refresh, these features will be activated following the precedence score in ascending order.

Below is an example, similar to the previous one, of how to define numerical precedence to declared features in feature_model_declaration.rb. Here the activation order will be Document > Picture > Vocal > Video. The feature Document is activated first because its precedence score will be defined as 0 by default. The notation is the same as the DFS order except that here you can add a precedence score.

```
feature :@picture, 'Picture', [:MessageModel], 3
feature :@document, 'Document', [:MessageModel]
feature :@vocal, 'Vocal', [:MessageModel], 4
feature :@video, 'Video', [:MessageModel], 5
@file_type.relation :Mandatory, [@document]
@file_type.relation :Optional, [@picture, @vocal, @video]
```

Listing 5.2: Feature model precedence example

The hierarchical order may not be respected here as the developer can give a child precedence over its parent. To counter this problem, if a child has precedence over its parent, an error is thrown and tells the developer which child has an incorrect precedence score. To check this, we explore the whole tree in DFS order, and for each feature, we verify that its precedence score is bigger (therefore the precedence is lower) than its parent, and if it is not the case, we throw an error. Only checking this property for each feature's parent is enough, because this property is transitive and propagated in the whole model. If all nodes have a correct precedence score compared to their parent, then all nodes have a correct precedence score compared to all their ancestors, because it has already been checked for each parent-child relation.

In the case of precedence score equality, a tie-breaker is needed. The tie-breaker for the numerical precedence strategy is the DFS (hierarchical) order, which means that if two features have the same score, the feature that appears first in the DFS order will be activated first. This ensures that the hierarchical order is respected if all features have the same precedence score. The way we use DFS as a tie-breaker is simply by first adding all features in a list in DFS order and then applying a stable sort by ascending order of precedence score. Because the sort is stable, the order will follow the DFS order when all features have the same precedence. For

this reason, we chose to give all features with a precedence score of 0 by default. This way, the default behaviour of this strategy will be to activate features in DFS order unless developers define new precedence scores.

The sort used for this strategy is the quicksort implementation of Ruby 2.5.5 called by the method *sort_by*. Ruby's implementation of quicksort is not stable therefore we made it stable by adding indexes to all elements. This allows sorting first on the precedence score and then using the indexes as a tie-breaker to make the sort stable. Our implementation of the stable sort is shown in figure 5.3 with l being the list of features to sort.

```
l.sort_by.with_index{|(aoe), i| [aoe.entity.precedence,i]}
```

Listing 5.3: stable sort in Numerical precedence strategy

Ruby 2.5.5 default sort algorithm is quicksort which has a mean time complexity of $\mathcal{O}(n \log n)$ and a worst case of $\mathcal{O}(n^2)$. For the space complexity, the ruby implementation has a $\mathcal{O}(\log n)$ mean memory usage and $\mathcal{O}(n)$ for the worst case. The modifications we added to make it stable do not change the time or space complexity because we only added indexes to the elements of the sorted list. This gives us a final time complexity of $\mathcal{O}(n \log n)$ in average and $\mathcal{O}(n^2)$ for the numerical precedence strategy.

The numerical precedence strategy guarantees both no sequential conflicts occurring and hierarchical order as long as the developer does not give higher precedence to the children compared to their parent feature which would throw an exception.

5.5 Relative precedence

In this section, we will first give a lexicon of the new notations and words we will use in this section. Second, we will explain how the precedence model is built based on the feature model and precedence relations. Finally, we will present the different algorithms used for implementing this strategy and we will prove their time complexity.

5.5.1 Lexicon

We will define **redundant relation** as follow:
A redundant relation is a relation that defines f1 > f2 where f1 is an ancestor of f2 or equivalently, where f2 is a successor of f1.
We name such a relation redundant because the system already ensures the hierarchical order by default. Redundant relations do not add any information to the precedence model and have no influence on the activation order.

When talking about **complexity**, we will name **N** the number of features defined in the feature model, **M** the maximum number of relations defined by the developers and **M'** the maximum number of non-redundant relations defined by the developers. Finally **d** represents the depth of the feature model.

We take M and M' as maximum values because when addressing a complexity, we need to look at the worst case. The worst case in the following algorithms will always correspond to M and M' having the maximum number of relations possible. This number is directly linked to the size of N and will be explained in details in Section 5.5.3

5.5.2 Building the Precedence Model

When using the relative precedence strategy, developers can choose the feature activation order for their adaptative systems. To do this, they define relations between the features. All the relations are then used to build a Precedence Model. It can be represented using a Hasse Diagram depicting which feature takes precedence over which features. It can also be represented as a Directed Acyclic Graph where each node is a feature and each edge is a precedence relation. This model allows the system to establish the list of all features sorted in their correct activation order. Once this list is built, it becomes easy to activate and deactivate the features in the correct order, as it only requires looking up the list to know which feature should be activated next.

As the model is built using relations, the time complexity to build it depends on the number M of relations defined by the developer. For example, considering features A,B,C,D,E,F and relations as follows:

- D > C
- D > F
- G > E
- F > G

In our extension of RubyCOP, such relations can be defined as follows :

```
def _define_precedence()
    add_precedence_relation("D", "C")
    add_precedence_relation("D", "F")
    add_precedence_relation("G", "E")
    add_precedence_relation("F", "G")
end
```

Listing 5.4: Defining precedence relation

We can represent the relation model by using a Hasse diagram in figure 5.1 and the features with a feature model in figure 5.2 :

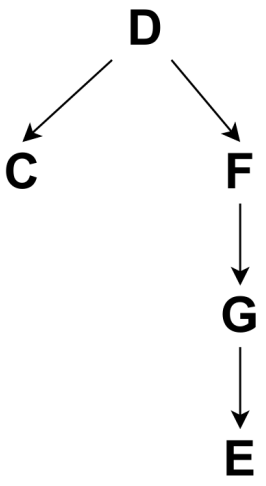

Figure 5.1: Hasse diagram

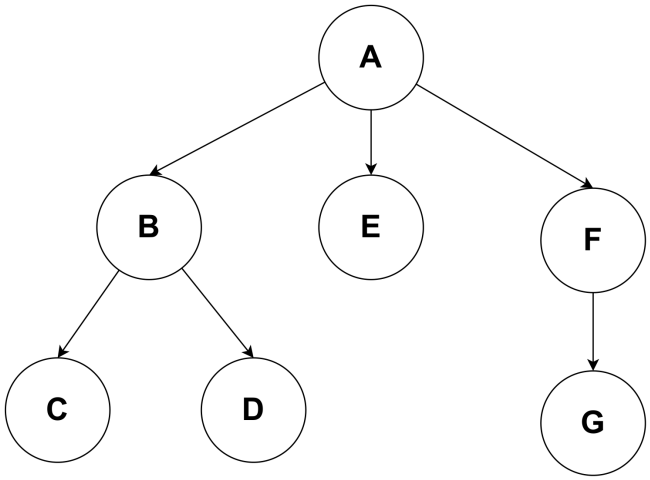

Figure 5.2: feature model

We will be using those graphs to create a DAG that will help us set up the activation order list. We will see how this algorithm is working in Listing 5.6. After merging the Hasse diagram and the feature model. We obtain the DAG as shown in figure 5.3 :

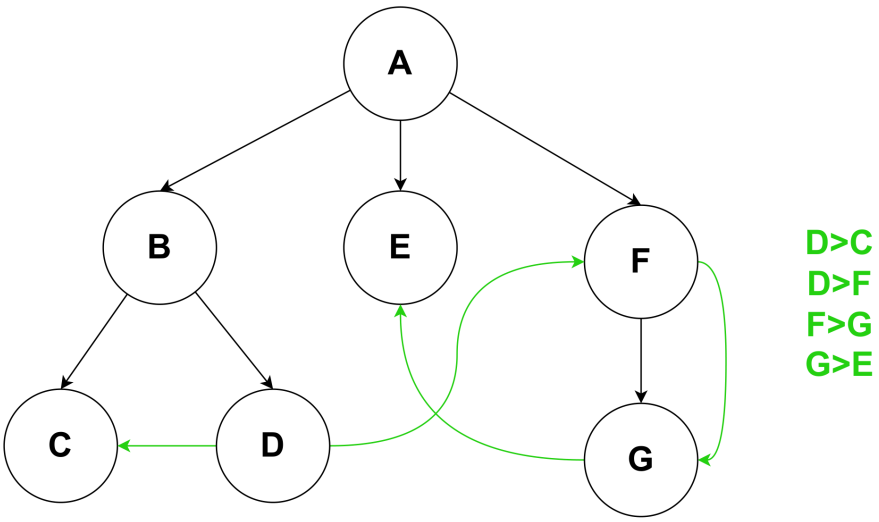

Figure 5.3: merged graph, DAG

5.5.3 Algorithm #1: Remove redundant relations

The first algorithm we will explain is the one removing redundant relations. If developers add relations that are already implied by the hierarchical order, the algorithm will remove them. It acts like a filter to optimise the 2nd algorithm that we will explain in section 5.5.4.

In listing 5.5, you will find the pseudo-code followed by an explanation of its complexity. Then in figure 5.4 an example of its application.

The algorithm to remove redundant relations presented in listing 5.5 is straightforward. It simply loops through each relation (f1, f2) and checks if f1 is an ancestor of f2. If it is the case then the relation is redundant and should be removed from the relation set.

```
remove_redundant_relations(relations):
    relation_set = ∅
    for each relations (f1,f2): #where f1 > f2
        ancestors_f2 = get_ancestors(f2)
        if not (f1 ∈ ancestors_f2) :
            relation_set.add(Relation(f1,f2))
    return relation_set
```

Listing 5.5: Remove redundant relations pseudocode

For the complexity of the algorithm presented in listing 5.5, we have a time complexity of $\mathcal{O}(M * d)$. M corresponds to the loop on each relation because all relations are explored. Inside this loop, we call get_ancestors(f2). Getting all ancestors cost $\mathcal{O}(d)$ because any feature cannot have more ancestors than the depth of the feature model. That is how we get a complexity of $\mathcal{O}(M * d)$ and the resulting set will be a set of size M' because we removed the redundant relations.

In figure 5.4, we can see that we are removing the relation $F > G$. This is because the hierarchical order already implies that F needs to be activated before G. Thanks to the hierarchical property, we can remove the redundant relation.

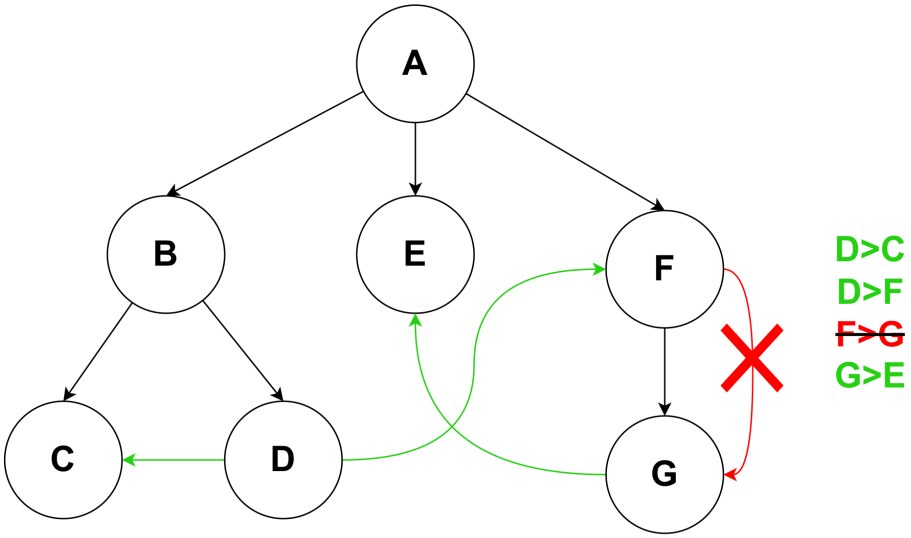

Figure 5.4: DAG with the removal of a redundant relation

Size of M and M'

The maximum number of relations that the developers can define is M. To prove how big M can be compared to N, we first need to understand our relation model. Relations must respect strict partial order set rules which are:

- Irreflexivity : a>a is impossible
- Asymmetry : if a>b then b>a is impossible
- Transitivity : if a>b and b>c then a>c

This means we can define N-1 relations for the first feature. To respect the asymmetry, We can define N-2 relations for the second feature and so forth. This follows an arithmetic progression :
$M = TotalRelation = (N-1) + (N-2) + (N-3) + ... + 2 + 1 + 0$. This can be represented as the sum :

$$M = \sum_{n=0}^{N-1} N = N(N-1)/2.$$

$$M = (N^2 - N)/2$$

Following this, the relations maximum space complexity is $\mathcal{O}(N^2)$.
In all of those relations, some of them are redundant when following the hierarchical

order. Taking a look at the figure 5.4 the Relation $F > G$ is redundant, therefore we can remove it without losing information.

Let us define R as the maximal number of redundant relations we can remove. As a reminder, a relation is redundant if it corresponds to a hierarchical relation of the feature model. If a feature has precedence over one of its successors, the relation is redundant. We can then compute the maximum number of redundant relations that is:

$$R = \sum_{n=0}^{N-1} depth(n)$$

Depth(n) corresponds to the depth at the node n. The value of R is dependent on the shape of the model and therefore cannot be simplified.

The final maximum number of non-redundant relations will be defined by: $M' = M - R$

5.5.4 Algorithm #2: Create activation order

Two graphs are needed for this algorithm. The features model and the precedence-relation model. As we said in the section 5.5.2, we can merge them into a DAG. This makes the implementation of the algorithm easier because we can get the activation order by using a variant of the Topological sort. The implementation of our topological sort is inspired by Kahn's algorithm [13] and works as follow:

- 1. Pop the first element of the heap and add it to the activation order list
- 2. Add to the heap all its children that have no other parents.
- 3. Adding the node to a visited set (similar to removing the node from the graph)
- 4. Repeat until the heap is empty

Since our implementation is more specific, we will explain it now in more detail by looking at its pseudocode presented in listing 5.6:

First, we use a min binary heap that sorts all added features by the index of DFS order. This allows us to be sure that the hierarchical order is maintained and that DFS order is used as a tie-breaker. Because whenever we pop a feature out of the heap, we are assured that it has the earliest DFS order possible between the available features.

When no relations (or only redundant relations) are defined, the algorithm will simply act as a DFS, popping all the features in DFS order.
Otherwise, the algorithm builds the activation order list incrementally, adding features one by one simply by popping the next element in the heap. To do that, it first adds the root feature to the heap and pops it, and then adds its children that have no other parents in the DAG. We will call such nodes orphans. Then the feature node is added to a set of visited nodes and we repeat the whole step for the next popped node. Adding the popped node to the visited set is similar to removing it from the graph. This is because the system checks if a node is an orphan by checking if all its parents are included in the visited set.

This algorithm is also able to detect loops between the precedence relations. If at the end of the execution, there are some features missing from the activation order, it means that a cycle exists inside the graph, which signifies that the developers have defined impossible relations such as B > C and C > B. A graph with a cycle is not a DAG anymore and no cycle should exist inside it. That is why when it happens, the system will raise an error, warning the developers to change their relations to remove the cycle in the graph.

```
create_activation_order():
    activation_order = []
    min_heap = MinPQ.new()
    min_heap.add(RootFeature)
    visited = ∅

    until min_heap.not empty:
        current_feature = min_heap.pop()
        activation_order_list.add(current_feature)
        visited.add(activated_feature)

        for child in current_feature.children:
            orphan = child.parents ⊆ visited
            if orphan?: #no parents left
                min_heap.add(child)

    return activation_order_list
```

Listing 5.6: Create activation order pseudocode

Complexity

When there is no cycle in the DAG, the algorithm first iterates on all the features inside the heap for a cost of $\theta(N)$.

Inside this loop, we pop the first element of the heap into the activation order list which takes $\mathcal{O}(logN)$ with the implementation of our heap.

With this added feature, we check for each of its children if those children can be activated. The cost of this operation is proportional to the number of children of the added feature. It is $\mathcal{O}(N)$ because one feature could be the parent of all other features in the model except of its ancestors.

Note that only orphan features can be activated. To check if it is the case, all its parents must be already in the visited set. This condition costs us a time complexity of $\mathcal{O}(N)$ with N being the total number of parents. If the feature is an orphan, we can add it to the heap.

Adding up all the operations, we get a final time complexity of $\mathcal{O}(N^3 * logN)$. This is the worst-case complexity that happens when developers define the maximal amount of non-redundant relation possible M' as mentioned in the section 5.5.3

We explore a total N features and we go through each of their parents and children which is related to M'. The more relations are defined, the more children and parents the features will have. Following what we said in the section 5.5.3, $\mathcal{O}(M')$ $= \mathcal{O}(N^2)$, we can define the complexity of the algorithm in Listing 5.6 as being $\mathcal{O}(N * M' * logN)$ (if M' is not equal to 0). This show again, that the developers should not define too many useless relations because it increases the time complexity of the algorithms.

The best case possible is when the developers do not define any relations or only redundant relations. Because we explore N nodes and add them to the heap N times. We only visit one child relation and only one parent relation. The comparison between two sets when checking if a node is an orphan becomes a comparison of one element to a set costing $\mathcal{O}(1)$ instead of $\mathcal{O}(N)$. In this case, the time complexity of the algorithm presented in Listing 5.6 becomes $\mathcal{O}(N^2log(N))$. The $log(N)$ is the cost when we add an element to the heap. Note that If no relations are defined, the algorithm is not needed and the DFS order will be used for the activation order. The Depth First Seach algorithm used for the activation order is done with a time complexity of $\theta(N)$.

In reality, the time complexity should never attain $\mathcal{O}(N^3 log(N))$ because it only happens if the developers create M' non-redundant relations. Knowing that only N-1 relations are enough to create all possible activation orders, each relation created above this number can be unnecessary. The only reason why it is allowed for the user to define more than N-1 relations is to promote expressivity. This way, the developers do not have to create the perfect activation order in advance and can just define any relations that seem important for them even though they are unnecessary or if it could be done with fewer relations.

The use of this strategy should therefore have an average complexity of $\mathcal{O}(N^2 log(N))$, it corresponds to the case where the developers define N-1 non-redundant relations (M'= N-1).

5.6 Conclusion

In this chapter, we saw the implementation of our three strategies and their time complexity. All those strategies resolve incompatibility conflicts and can be used in RubyCOP.

When it comes to tie-breakers, the DFS order seems to be the most optimal choice as it uses less memory than the BFS order. It is also more faithful to the definition of the feature model: Each feature is refined by their children. It makes more sense to activate one parent node and its children consecutively and so on than activating first all the features, and then their children layer by layer.

The comparison of those strategies based on their strengths and weaknesses and how they work on concrete examples will be discussed in the next chapter about validation.

Chapter 6

Validation

In the previous section, we have seen the implementation of our different strategies and how developers can use them. The main purpose of this section is to validate the different strategies we have implemented. Those strategies are supposed to solve sequential conflicts as we have seen in the previous chapters. We are going to see if there are limitations to those strategies and in which case they are the most effective. To validate our strategies, we will use our running example of the smart messaging system to make a case study.
First of all, we will give a complete definition of it. Then we will proceed to the validation by presenting different examples and how they are handled by the different strategies. Finally, we will explain the design choice of using DFS instead of BFS, as well as the threats to validity of our thesis.

6.1 Case study of the running example

As we said in the introduction above, we are going to use the smart messaging system we developed and already quickly presented as our running example in section 4.1. By using the smart messaging system, the user is allowed to create chats with other users and send messages, pictures and locations into the chatbox. The system is adaptive depending on the different contexts the application is currently running on. For example, if the user's phone is equipped with a camera, then he will be able to send a picture or a video in the chat, if a GPS is detected, the user will be able to share its location, they can send a voice message if there is a microphone and if the network quality is bad, all the message sent will be compressed to reduce the size of the communication. The context-feature model of our smart messaging system is shown in figure 6.1 as well as the lexicon in table 6.1 and 6.2 which will define the features and the contexts. We did not explicitly define all of them for simplicity such as the feature Message which is already self-explanatory.

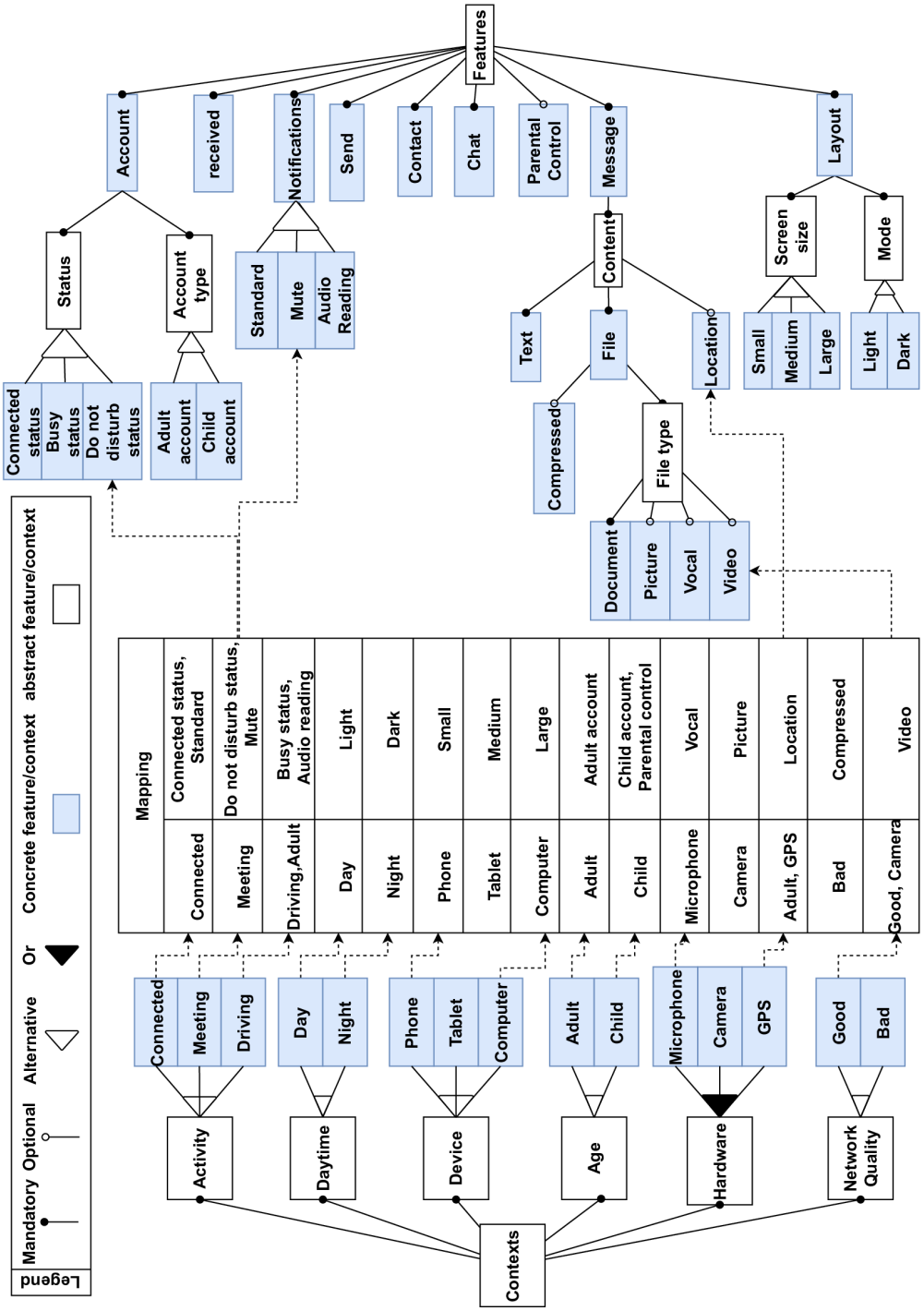

Figure 6.1: Context-feature model of the smart-messaging system

Case Study's lexicon

Feature	Definition
contacts	A contact is another user who you can start chatting with. This feature includes all features related to managing the list of contacts like adding, removing contacts, displaying the contacts and searching among them. Note that the added contact does not have to accept any invitation (the relation is unidirectional so once you add a contact, you can immediately start chatting with him).
Message - Content	A message sent between two users can have multiple elements in its content. It can have a text, a location and a file.
Location	This type of content allows showing in the message the location where the user is currently located. It is useful when the user wants to show his friends where he is travelling for example.
File	This type of content can be a vocal (audio message), a document, a picture or a video. Those types of files can be compressed if the system detects the user has a bad network quality.
Account	To use the app, every user needs to register an account. It contains a username, profile picture, bio, status, list of contacts, list of notifications, information about the age of the user, list of chats the user is participating to.
Status	Status is displayed on the profile. It can be "connected" when the user is available, "busy" while driving or "do not disturb" when he does not want to be disturbed (e.g. while in a meeting). It's useful to know if a contact is busy or not before calling them so we don't waste time trying to reach him if he is not available.
AccountType	AccountType differentiates the type of account depending on the user's age. Two types of accounts are possible: Child and Parents. Here, parents may want to manage their children's account
Chat	A chat is a conversation between two or more users. Every member of the chat can send messages on it and read all messages sent by other members. Each user can have multiple chats.
Parental control	A parental control that allows parents to read messages received by their child. This feature is useful for a messaging application because it allows parents to have security over their children so they do not talk with strangers or send/receive suspect messages.
Notification	It is the textual notification triggered when the user receives new messages from chats.
Notification - Sound	It is simply the sound produced when the user receives a notification. It can be in "standard" mode when it behaves normally (the ringtone produces a sound), in "mute" mode when the ringtone is muted or in "audio reading" when the received message is read loud by the application. This feature is useful for a messaging application because it gives an audible warning when a message is received.
Layout - Mode	The display mode can be either the dark mode or the light mode. This is useful for an application because it allows to user to have an optimal display whether it is day or night.

Table 6.1: Lexicon defining features

Context	Definition
Activity	The activity is what the user is doing. He can be connected, driving or in a meeting (a meeting is an example of an activity where the user does not want to be disturbed).
Device	Indicates the kind of device used by the user.
Age	Indicates if the user using the application is a child or an adult.
Hardware	Indicates the hardware supported by the device of the user.
Network quality	Indicates if the quality of the network the user is connected to is poor.

Table 6.2: Lexicon defining Context

6.2 Cases introduction

In the following cases, we consider that the following contexts are activated: [Connected, Day, Phone, Adult, Microphone, Camera, Gps, Good]. Therefore, the following features are activated (in DFS order): [Account, Connected status, Adult account, Received, Notifications, Standard, Send, Contact, Chat, Message, Text, File, Document, Picture, Vocal, Video, Location, Layout, Small, Light]. We also consider that they stay activated between each message of the smart messaging system. Only the order in which they were activated changes (to emulate a sequential conflict).

Case 1: Sequential conflict

In this first case, we are going to show a direct example of sequential conflict and which solutions can be used to solve this situation.

Contextualisation

Sam is a music producer, he is using the smart messaging system to send his production to his client Bob. In the morning, Sam sends his current work to Bob, his client. This message is containing, the text of the message, a pdf file, a picture, an audio file and the location. In the afternoon, he is sending the second rush to his client with the same files as usual.

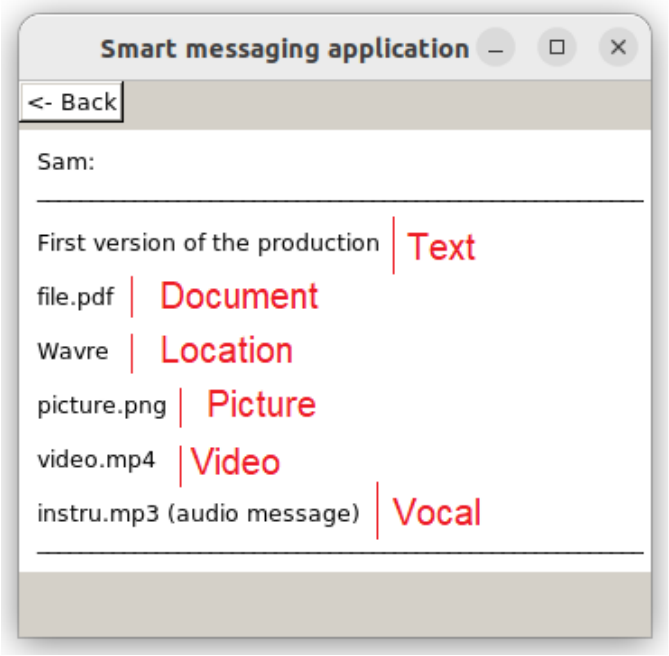

Figure 6.2: First message

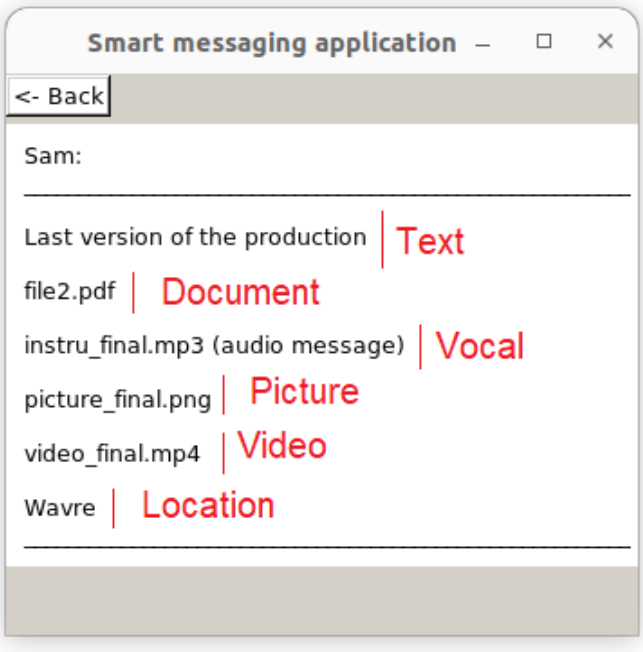

Figure 6.3: Second message

In this example, the activated contexts and feature (configuration) stay the same between both messages. We consider that the user closed the application and reopened it later. As you can notice between the figures 6.2 and 6.3 the disposition is different, despite, the configuration being exactly the same. This is a concrete example of a sequential conflict. The cause is that when closing and reopening the messaging application, the contexts were activated in a different order than the first time. This order had an impact on the behaviour (visually) of the messaging system, therefore creating a sequential conflict.

To solve this error, we can use the different strategies we have implemented. For example, we could use the basic DFS order strategy to solve this:

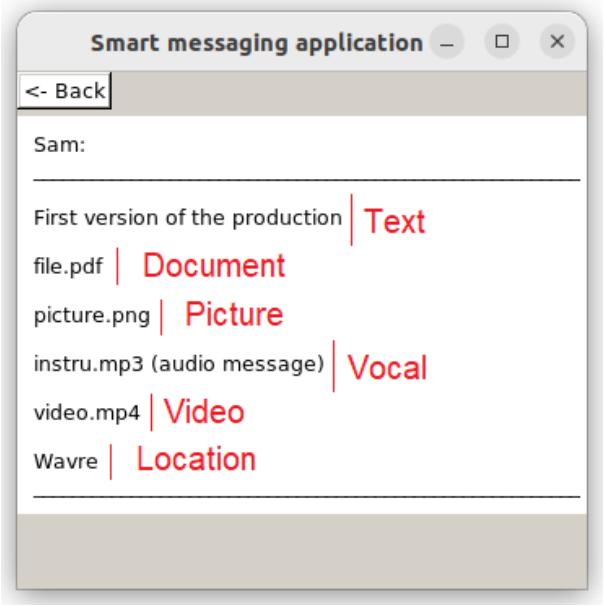

Figure 6.4: First message in DFS order

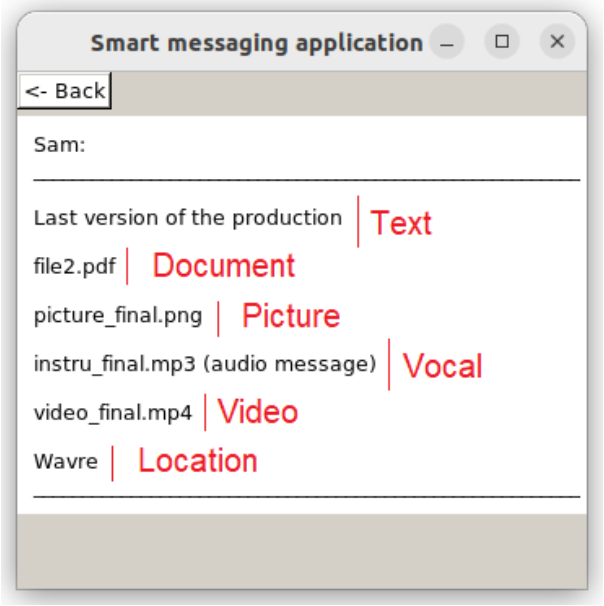

Figure 6.5: Second message in DFS order

After using the DFS order strategy we can notice in figures 6.4 and 6.5 we can see that, even if the context activation order is different, the final execution will remain the same. Our DFS strategy resolves the problem of sequential conflict in this example.
Both Numerical and Relative precedence strategies can be used to resolve the sequential conflicts of this first case. No precedence has to be defined because the only requirement here is to stop the conflicts. If no precedence is defined, then both strategies will act as a DFS order and will produce the same output as in figures 6.4 and 6.5. But even when developers define precedence as in Case 2 and Case 3, both strategies still resolve the sequential conflicts. Therefore, all strategies we implemented resolve sequential conflicts.

Case 2: Specific order of activation

We have seen that using the DFS order, based on the feature model, is enough to resolve sequential conflict. The first case was simplistic and did not show the need for developers to change the feature activation order. If developers want to change the DFS order, they have two solutions. They could change their model, and therefore change the DFS order. But they can also use our numerical or

relative precedence strategies. Those strategies enable developers to personalise the activation order.

Contextualisation

For example, a team of developers are implementing a version of the smart messaging system where sending images is the most important feature. When sending a message to another contact, we want the image to be displayed first. In the basic smart messaging system, the Text feature comes in first position in the DFS order, thus the Text will be displayed first.
If developers change the feature model in Figure 5.2, they will have to swap the Text feature and the File feature.

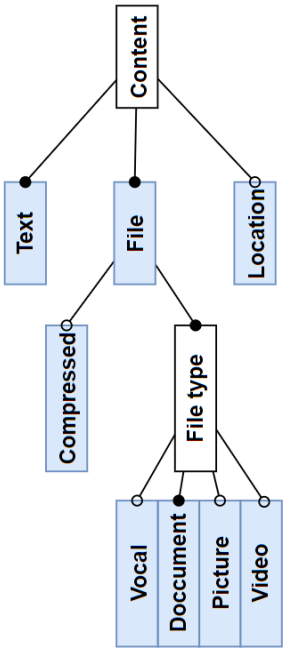

Figure 6.6: initial model

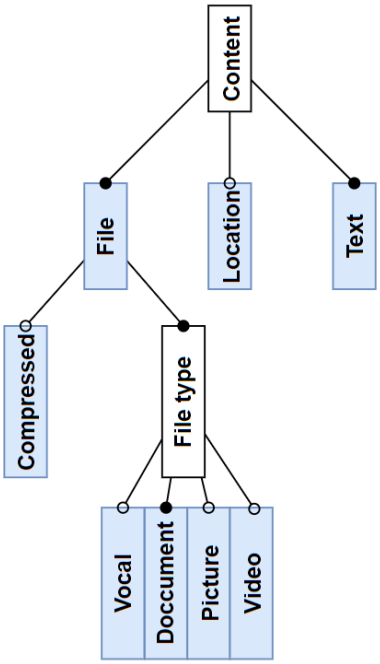

Figure 6.7: changed model

But instead of changing the feature model, the developers can also use the numerical precedence strategy as follow :

```
def _define_message_content()
    feature :@text, 'Text', [:MessageModel], 3
    feature :@location, 'Location', [:MessageModel], 2
    abstract_feature :@file, 'File'
    _define_file()
```

```
    @content.relation :Mandatory, [@text, @file]
    @content.relation :Optional, [@location]
end

def _define_file_types()
    feature :@picture, 'Picture', [:MessageModel], 0
    feature :@document, 'Document', [:MessageModel], 1
    feature :@vocal, 'Vocal', [:MessageModel], 1
    feature :@video, 'Video', [:MessageModel], 1
    @file_type.relation :Mandatory, [@document]
    @file_type.relation :Optional, [@picture, @vocal, @video]
end
```

Listing 6.1: Numerical precedence declaration

They can also use the relative precedence strategy :

```
def _belgian_app_prio()
    add_precedence_relation("Picture", "Location") #Picture > Location
    add_precedence_relation("Location", "Text")    #Location > Text
end
```

Listing 6.2: relative precedence declaration

When sending a message with a picture and the location, the message will now be displayed as follows :

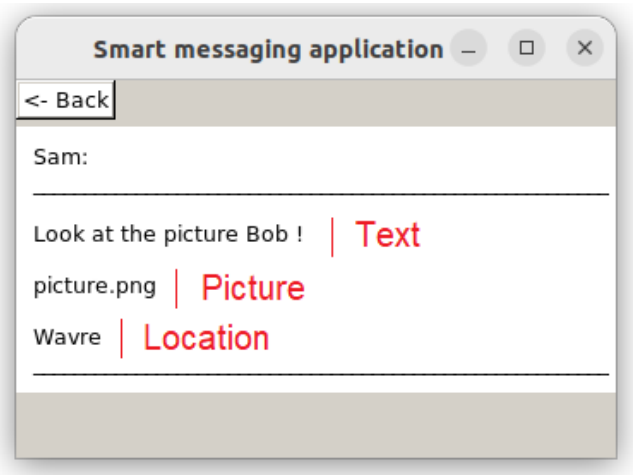

Figure 6.8: case 2: initial order

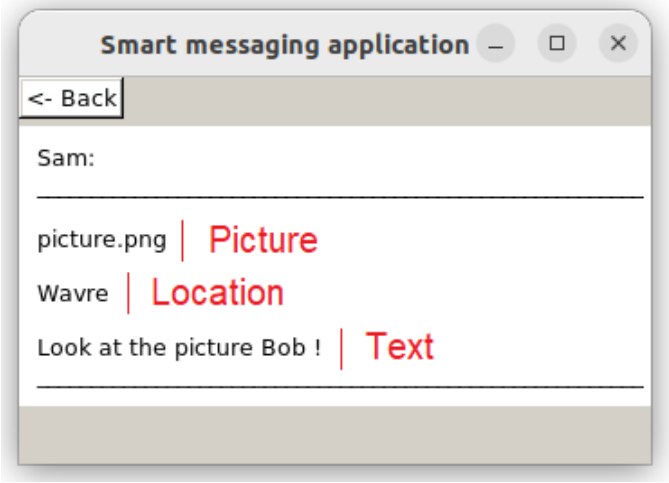

Figure 6.9: case 2: final order

In figures 6.8 and 6.9, we can see that by changing the model or by using the strategies, the final order of the application corresponds to the needs of the developers.

Case 3: Specific order of activation with interleavings

Our three strategies can be used to change the order of activation to answer the developers' needs. However, we are going to present a more complex case where developers want an order of activation with non-sibling feature interleavings.

Contextualisation

As an example, we could imagine the Belgian government asked developers to create an alert messaging system to send alert notifications in case of emergencies. Developers want to reuse the smart messaging system as a basis for the new alert system to reduce their development costs. But, the government wants the application to send the alert message with a specific layout: The image with the map has to appear in the first position, followed by the location, then the pdf with the instructions and finally the text in the last position. Such configuration is not achievable by using the DFS strategy. Developers could change the whole model but the workload involved in changing the model is considerable. The reason is that the developers would have to change the hierarchical structure of their model in order to do interleavings of features with a DFS. DFS explore all children of a feature before exploring the next features, Taking our case example, the DFS strategy cannot explore one child of File (Picture) then explore Location (outside of File) and go back to get another child of File (Document).

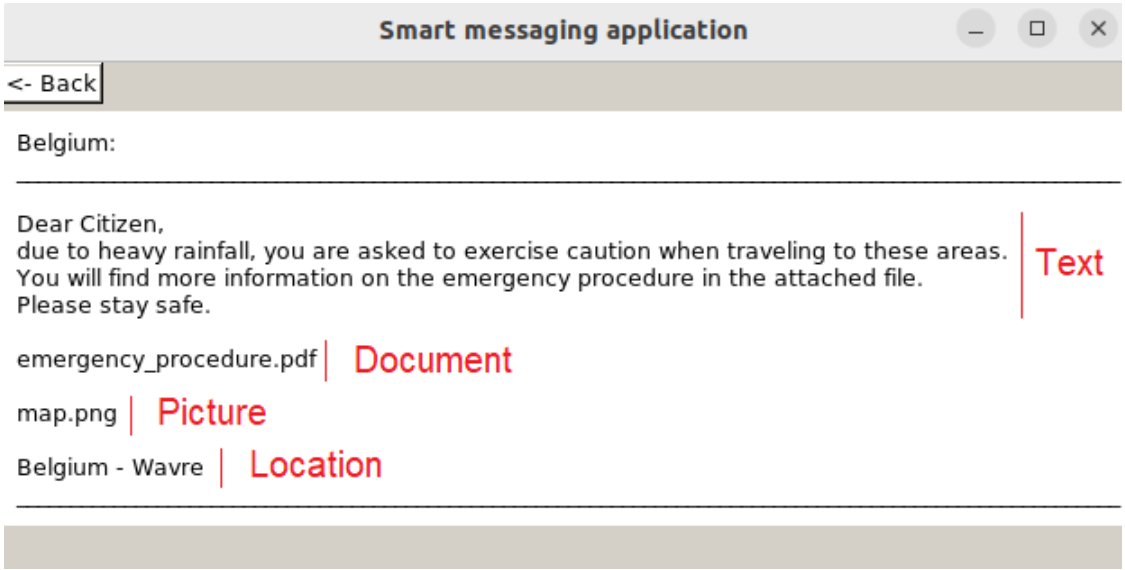

Figure 6.10: Case 3 : DFS order

As you can see in figure 6.10, using the DFS order does not output the wanted order of activation.

One solution for this case is to use the numerical or relational precedence strategy to personalise the feature activation order that fits the needs of the government. By using the numerical strategy, developers could define the precedences as follows :

```
def _define_message_content()
    feature :@text, 'Text', [:MessageModel], 10
    feature :@location, 'Location', [:MessageModel], 2
    abstract_feature :@file, 'File'
    _define_file()
    @content.relation :Mandatory, [@text, @file]
    @content.relation :Optional, [@location]
end

def _define_file_types()
    feature :@picture, 'Picture', [:MessageModel], 0
    feature :@document, 'Document', [:MessageModel], 3
    feature :@vocal, 'Vocal', [:MessageModel], 0
    feature :@video, 'Video', [:MessageModel], 0
    @file_type.relation :Mandatory, [@document]
    @file_type.relation :Optional, [@picture, @vocal, @video]
end
```

Listing 6.3: Numerical precedence declaration

Another solution is by using the relation precedence strategy :

```
def _belgian_app_prio()
    add_precedence_relation("Picture", "Location") #Picture > Location
    add_precedence_relation("Document", "Text")    #Document > Text
    add_precedence_relation("Video", "Document")   #Video > Document
    add_precedence_relation("Location", "Document")#Location > Document
end
```

Listing 6.4: relative precedence declaration

With this new personalised model, the execution of the application will satisfy the needs of the government.

Figure 6.11: Case 3: final order

By using our strategy, developers are able to personalise the order of activation they want. They also need to be aware that not all activation orders are possible. Indeed as we discussed in chapter 5, activation order must respect constraints.

Summary of the case study

We created a summary of the case study in table 6.3. This table shows for each strategy, how they perform on the different cases. The three properties correspond to the three cases, respectively: Case 1 -> Sequential conflicts, Case 2 -> Specific order and Case 3 -> Interleavings.

Properties/Strategy	DFS	Numerical	Relative
Seq. conflicts	+	+	+
Specific order	-/+[1]	+	+
Interleavings	-	+	+

Table 6.3: Comparison of strategies with the cases

[1]DFS cannot change the activation order (-), unless developers change the model (+)

6.3 Analysis of the strategies

In this section, we will take a step back and analyse our strategies to evaluate their general usability.

DFS

The DFS strategy works amazingly as a tie-breaker. It is used for this purpose in both other strategies. The question we should ask ourselves is whether this strategy can be used alone or not. To answer this question, we can take a look at our case study. It shows that it resolves sequential conflicts, and allows specific activation orders (if changing the model) but cannot put interleavings in the activation order. Do developers often want interleaved feature activation orders? This depends on the type of program. Some programs could need several of them while others will not need any. It also depends on the structure of the feature model. For simpler models, this should not be a problem. The model can always be changed in order to adapt the activation order. Interchanging nodes among the same layer is easy, but changing nodes with different levels of layers is a tricky task and often time-consuming.

In terms of performance, this strategy has the best time complexity of all 3 strategies because it only has to explore the model once to find the activation order. It is also easy to use for developers because they do not have to define scores or relations. But they have to change the model when they want a specific activation order, even though this remains easy to do unless there are feature interleavings.

We recommend developers start using the DFS order strategy to understand strategies and how to control the activation order. Because this strategy is easy to apply, solve sequential conflicts and does not require any effort from developers unless they want specific activation order. Later, when building more complex programs, the developers could learn to use other strategies and change the activation order.

Numerical precedence

Another approach to resolve sequential conflicts is using Numerical precedence. The numerical precedence strategy is effective and has good complexity. as we explained in section 5.4.

The problem with this strategy as we have seen, is that it requires more work from the developers compared to the other strategies. It is the hardest strategy to

use and to understand, it lacks expressivity. Using numbers is harder to interpret compared to relations; Numbers are more abstract. When multiple developers use the numerical strategy. The problem is that those developers can have different definitions of precedence score: a feature with 10 as a score can be considered "high precedence" for one developer whereas this same score can be considered as "low precedence" for the other. This problem is bothersome when it comes to maintainability and can introduce confusion amount developers. They need to find a common ground to solve this issue.

We recommend using this strategy only for specific purposes such as debugging. Developers should be careful when using this strategy because adding one score to a non-leave feature requires adding compatible scores with all its successors. This strategy can also be used in applications where performance is extremely important because this strategy has a great time complexity. But developers have to agree on a general rule to define precedence scores.

Relative precedence

At first, the relative precedence strategy seems to be the perfect strategy. Indeed, it can resolve sequential conflicts, choose a specific order and even when there are interleavings. It is also expressive, the developers can easily understand it and use it. However, the main issue we have with this strategy is its complexity as it was explained in Section 5.5. The second issue is that developers have to be careful not to create cycles in the model with their relations.

We recommend using the relative strategy as a default strategy for developers that want to use an expressive strategy. The focus of this strategy is to be easy to use and to learn. It also allows enough customisation to have specific orders with interleavings. This strategy should not be used in systems where the time performance is important.

6.4 Common mistakes for developers

Using strategies in adaptive systems can be challenging for developers that are not accustomed to strategies. In this section, we enumerate the most common mistakes that can be made while using our implemented strategies.

We'll start explaining the **badly designed feature model** mistake. It can occur in any strategy, but mainly in the DFS strategy because it only uses the feature model. When designing their feature model, developers have to think in advance in which order they define the features in the model. Especially if they intend to use the DFS strategy to resolve sequential conflicts. Because the features will be activated in the order in which they appear in the model. A badly designed feature model will create errors in the application. For example, when activated features are dependent on other features that are not activated yet. But many other errors can occur because of that.

The **child>parent precedence** mistake can occur in both Numerical and Relative precedence strategies. It is equivalent to not respecting the hierarchical order. When doing this mistake, an exception is thrown to avoid the system having unexpected behaviours. It corresponds to giving precedence to a feature over one of its ancestors.

The **precedence cycle** corresponds to having a cycle inside the precedence model. This can only occur in the relative strategy because it uses a topological sort which requires no cycle in the relations. A cycle can be as direct as A > B and B > A. Or it can be by transitivity such as A > B, B > C and C > A. This results in an exception because no topological order respecting the relations can be found.

The **undefined score** mistake is unique to the Numerical strategy. The precedence score of all features starts at 0. It is important when defining a score for a non-leave feature, to define all scores of its successors. Otherwise, we have a Child>Parent precedence. Avoiding this requires more work from the developer.
An example is shown in Figure 6.12. The developers add a precedence score of 6 for the layout but they must now configure all Layout's children's scores.
It would then require defining 5 new scores for the 5 non-abstract successors of the feature Layout.

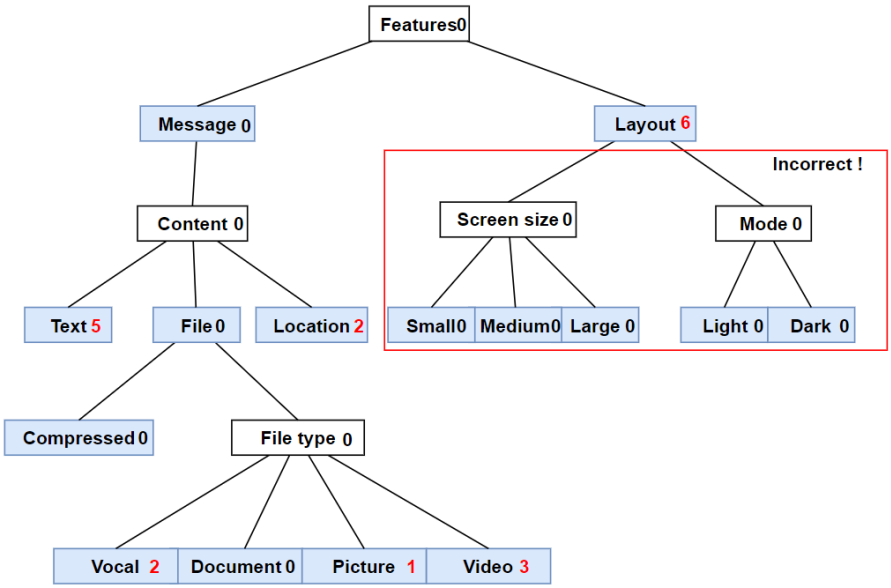

Figure 6.12: Undefined score mistake

6.5 Summary of the strategies

We created a summary of our strategies in table 6.4 by extending the previous table 6.3. This table shows for each strategy, how they perform on different properties. The three first properties correspond to the three cases, respectively: Case 1 -> Sequential conflicts, Case 2 -> Specific order and Case 3 -> Interleavings. The expressivity, performance and common mistakes were added and are discussed in section 6.3 and 6.4. The expressivity corresponds to how a strategy is easy to understand and to use for the developers.

Properties/Strategy	DFS	Numerical	Relative
Seq. conflicts	+	+	+
Specific order	-/+[1]	+	+
Interleavings	-	+	+
Expressivity	+	-	+
Performance	+	+	-
Common mistakes	Badly designed model	Child>parent, Undef. score	Child>parent, Cycle

Table 6.4: Comparison of strategies with their priorities

[1]DFS cannot change the activation order (-), unless developers change the model (+)

6.6 Threats to validity

Because we implemented, tested and validated the strategies ourselves, it is important to note that this thesis may be biased. We tried to be as neutral as possible when comparing and analysing the strategies but this is not sufficient to ensure the perfect validity of our work. External points of view are necessary to agree with our claims.

Since the main topic of this thesis was about exploring and testing different strategies, we only made one single case study for the sequential conflicts. The choice behind that was to put all our efforts into one case study to make it as complete and correct as possible. As well as giving multiple examples for the same case study. However, adding additional case studies would improve the validity of our work.

6.7 Conclusion

The three strategies resolve sequential conflicts and have their strengths and weaknesses.

The DFS order strategy is fast, easy to use and to understand, but lacks adaptability. Indeed, whenever developers want to make a change in the activation order, they have to change their model. This can be simple for sibling features but becomes complex when wanting to interleave non-sibling features.

The numerical precedence strategy has good time complexity but is hard to use for the developer. Managing adaptive systems with complex feature models is a tricky task. Declaring numerical precedence can be a very difficult task for the developer to handle because he must cope with a lot of numbers to prioritize features. The developer must also take care to respect the hierarchical order so as to avoid incorrect behaviour or even system crashes.

The relative precedence strategy, even though it is more computer intensive than its counterpart, is fixing a lot of issues the numerical precedence strategy has. With relations, developers have common ground by default. Feature A has precedence over feature B (A $>$ B), there is no in-between. It is also easier to express complex order of activation with siblings and also with features that are distant from each other in the model. This is possible thanks to the fact that relations are transitive and will respect the hierarchical order without the help of the developer. Those are the reason why the relative precedence strategy has better expressivity. But

the lack of performance can be a major drawback in some applications.

As for now, our extension of RubyCOP is able to manage sequential conflicts by using any of the 3 strategies we implemented. Those strategies can also be used to choose a specific feature activation order in certain conditions.

Chapter 7

Future work

As we have seen throughout this thesis, conflicts can occur in many different ways. We have provided leads and solutions for resolving and managing those conflicts. In this chapter, we are going to explore solutions to enhance our strategy mechanism, improve the expressivity of our solutions and other ideas that could lead to future work on the FBCOP paradigm and conflicts in COP more generally.

7.1 Strategies

We have explored a multitude of strategies in the chapter 3 and 4. We can always add new strategies if the developer has specific needs.

Conditional strategy

One way to improve strategies is to add conditional clauses they need to respect. By adding conditions, the system could have multiple orders of activation. Let us say that, in our messaging system we want to display the text before the image, the files and so on. let us say that we have a context corresponding to the user's preferences. The user could be dyslexic or with visions problem, making him want to have another display order. It could be, for example, the vocal file, the picture and then the text. By adding conditions to those specific contexts we could create different activation orders that will directly affect the display of the messaging system.

Adding conditions to our strategies could also be part of the incompatibility conflicts. If we take our case study Fire versus Flood in section 3.4, we can add a condition to choose which feature is the most important and needs to be activated. An example could be if, in a smart home system, there are contexts that corre-

spond to the insurance of the house owner. The insurance could be fire insurance, water damage, natural disaster, etc. Depending on those contexts, we could make the choice to let the fire burn instead of flooding it and vice versa. Conditional strategies could be a great addition to our work and it would enable developers greater control over the strategies they use by adding conditions.

Precedence list strategy

This strategy can be similar to the relative precedence strategy. It consists in defining the order of activation of all the features before the execution of the system. By doing so, the system just has to follow the order of activation defined by the developers. This strategy is inspired by the AspectJ extension where they use a similar method [16]. This approach has positive and negative advantages. For the positive part, the performance when using this strategy is greatly improved. Since the order is already defined, we just have to follow it and deploy features one by one. Compared to the relative precedence strategy where we have to infer the final order from the relations defined by the developer. The negative aspect of this approach is the workload imposed on the developers. They have to explicitly define a precedence order between possibly independent features, compared to the relative strategy where they do not need to define relations between all features. It becomes also harder when adding a new feature to the precedence order with this strategy as you would need to put it in the right position between two other features. This means that it becomes increasingly harder to add new features to the precedence order compared to our relative precedence strategy where you can add relations independently of the existing relations as long as they are not incompatible with those.

Static analysis for valid order

In the current version of RubyCOP, we are doing a Dynamic Analysis. At run time, the system verifies that the relation the developers are defining is valid and allowed. The idea of doing static analysis and implementing it in RubyCOP could be a great extension of the framework. Developers will have feedback at design time on the different relations he is writing and see if one of them is incompatible. If a relation is detected for violating a static rule, IDE could highlight the concerned relation and notify that the corresponding relation is not a valid one. Static rules could be based on the existing feature model and enriched for each new relation added.

Implementation for incompatible conflicts

In this thesis, we created an extension of the RubyCOP framework that allows us to resolve sequential conflicts. The next step is now to implement different strategies for incompatibility conflicts. In section 3.3, we had a discussion on which strategy could be the most relevant to implement. We conclude that the prioritising strategy, the Loyal-Prompt, the Rollback, the user-driven and the user as a context were the most interesting strategies in terms of potential resolving power. In terms of implementation difficulties, the prioritising, loyal and prompt strategies seemed the most achievable. We explicitly implemented our strategies as an interface for sequential conflicts so that other strategies can be easily implemented following this interface remaining as maintainable as possible. When implementing strategies for incompatibility conflicts, a similar implementation should be used. In order to be effective, the strategies require a detection tool in order to be detected.

Combining Numerical and Relative precedence strategies

One solution to have advantages of both numerical and relative precedence strategies while removing their drawbacks is to combine them into one strategy. We could have a strategy that would work exactly like the relative strategy but instead of building the entire precedence Model for each execution, we could simply do it once, then store the resulting model in a text file in the shape of numerical priorities and then every time we run the program again, we would simply use numerical priorities. Those numerical priorities would be built by using the relative priorities. But by transforming the relative into numerical priorities, we can have the exact same output as using the relative strategy while keeping the great time complexity of the Numerical strategy. It would still take as much time to build the first model with the relative strategy but then we would not have to build it until it is changed again. The system will create a copy of the last precedence Model and every time we run the application, the system will compare both the old and new precedence Model and builds it only if it detects a change. But except while creating the application, the precedence Model should not change often, because once the developer has found out the best priorities for the features, he does not have to change them again unless he adds new features. New features are not created too often so we would keep the same precedence model without having to build it again and again with our new combined strategy, allowing us to save a great amount of time . This solution is the best compromise between both strategies.

7.2 Detection Tools

As we have seen in the section 3.2 detecting potential conflict is a hard task. We need detection tools in order to detect if features a potentially incompatible behaviour when they are activated together. Some detection mechanisms can be implemented into RubyCOP. An example we have seen in chapter 2.2 is to use reuse contract [24] and detect if features are entangled. When the system adapts itself and deploys a new feature to the source code, the system could dynamically detect that the new feature is causing a conflict and upon detection, we could apply the different strategies we discuss in section 3.3. By using a detection tool, we will be able to fully detect incompatible conflicts that the developer might not be aware of, and dynamically manage them.

Chapter 8

Conclusion

During this thesis, we explored the feature-interaction problems in context-oriented programming. We explain both types of conflicts: incompatibility and sequential. Then, we presented several strategies for both conflicts. We provided an implementation in the RubyCOP framework[9] and defined our 3 implemented strategies to resolve sequential conflicts and their underlying algorithms and mechanisms. Finally, we validated our approach and strategies for sequential conflicts by comparing the strategies and by looking at different case studies. Our approach in order to solve sequential conflicts is to let developers be able to personalize the activation order of the features. Once the developers define it, the order is fixed and cannot change if features are activated and deactivated therefore solving the issue of sequential conflict.

Contribution

- We proposed more precise definitions of the feature-interaction problems, providing a case study and concrete examples. We defined two types of conflicts that could occur in adaptive systems: sequential and incompatible conflicts and we defined when and how they occur.
- We analysed and defined a multitude of strategies for managing both types of conflicts. Strategies are the solutions we found in order to solve those conflicts. We also enhanced the design space proposed by Mens, Duhoux and Cardozo [21] allowing developers to precisely define new strategies.
- We provided an implementation of the strategies to solve sequential conflicts in RubyCOP. These strategies are a tool for developers to resolve sequential conflicts. But also, giving opportunities for developers to personalise the layout of their applications by modifying the order of activation of features.

And allowing them to change the way the application displays its element without altering the main code of the application.

- We finally validated our implementation by comparing our strategies together with our running example as a case study. This allowed us to show the sequential conflicts resolved and what could happen without our precedence strategies.
- We presented a case study to compare our 3 strategies and discussed their properties. We created a table summarising the 3 strategies and how they perform related to several properties. We also explained how and when they should be used.

Bibliography

[1] Kim Barrett et al. “A Monotonic Superclass Linearization for Dylan”. In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’96. San Jose, California, USA: Association for Computing Machinery, 1996, pp. 69–82.

[2] Nelly Bencomo et al. “Self-Explanation in Adaptive Systems”. In: Jan. 2012, pp. 157–166.

[3] Lodewijk Bergmans and Cristina Videira Lopes. “Aspect-Oriented Programming”. In: *Object-Oriented Technology ECOOP’99 Workshop Reader*. Ed. by Ana Moreira. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 288–313. ISBN: 978-3-540-46589-8.

[4] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* College Publications, 2006. Chap. 9.4. ISBN: 1904987176.

[5] Nicolás Cardozo, Kim Mens, and Siobhán Clarke. “Models for the Consistent Interaction of Adaptations in Self-Adaptive Systems”. In: Jan. 2017, pp. 307–348. ISBN: 978-3-319-74182-6.

[6] Nicolás Cardozo et al. “Context Petri Nets: Enabling Consistent Composition of Context-Dependent Behavior”. In: vol. 851. June 2012, pp. 156–170.

[7] R. Ducournau et al. “Proposal for a Monotonic Multiple Inheritance Linearization”. In: *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*. OOPSLA ’94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 164–175. ISBN: 0897916883.

[8] Benoît Duhoux, Kim Mens, and Bruno Dumas. “Implementation of a Feature-Based Context-Oriented Programming Language”. In: July 15, 2019, pp. 9–16. ISBN: 978-1-4503-6863-6.

[9] Benoıt Duhoux. “Feature-Based Context-Oriented Software Development”. PhD thesis. Université catholique de Louvain, Aug. 2022.

[10] Erich Gamma et al. “Elements of Reusable Object-Oriented Software”. In: *Design Patterns* (1994).

[11] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. “Context-Oriented Programming”. In: *Journal of Object Technology* 7.3 (2008). In collab. with Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Num Pages: 27 Number: 3 Place: Zürich Publisher: AITO, pp. 125–151. ISSN: 1660-1769.

[12] Florent Hivert and Nicolas Thiéry. “Controlling the C3 Super Class Linearization Algorithm for Large Hierarchies of Classes”. In: *Order* (July 2022), pp. 1–16.

[13] A. B. Kahn. “Topological Sorting of Large Networks”. In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782.

[14] Kyo C. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study:* Fort Belvoir, VA: Defense Technical Information Center, Nov. 1, 1990.

[15] Andy Kellens et al. “Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts”. In: *ECOOP 2006 – Object-Oriented Programming*. Ed. by Dave Thomas. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 501–525. ISBN: 978-3-540-35727-8.

[16] Gregor Kiczales et al. “An Overview of AspectJ”. In: *ECOOP 2001 — Object-Oriented Programming*. Ed. by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 327–354. ISBN: 978-3-540-45337-6.

[17] Gregor Kiczales et al. “Aspect-oriented programming”. In: *ECOOP’97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer. 1997, pp. 220–242.

[18] Paul Leger, Nicolás Cardozo, and Hidehiko Masuhara. “An expressive and modular layer activation mechanism for Context-Oriented Programming”. In: *Information and Software Technology* 156 (2023), p. 107132. ISSN: 0950-5849.

[19] J W Lloyd. “Foundations of Logoic Progorammin<:” in: (1984).

[20] Donna Malayeri and Jonathan Aldrich. “CZ: Multiple Inheritance without Diamonds”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pp. 21–40.

[21] Kim Mens, Benoît Duhoux, and Nicolás Cardozo. “Managing the Context Interaction Problem”. In: *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming*. Programming ’17. New York, NY, USA: Association for Computing Machinery, Apr. 3, 2017, pp. 1–6.

[22] Kim Mens et al. “Modeling and Managing Context-Aware Systems’ Variability”. In: *IEEE Software* 34.6 (2017), pp. 58–63.

[23] Ghan Bir Singh. “Single versus multiple inheritance in object oriented programming”. In: *OOPS Messenger* 6 (1995), pp. 30–39.

[24] Patrick Steyaert et al. “Reuse Contracts: Managing the Evolution of Reusable Assets”. In: *SIGPLAN Not.* 31.10 (Oct. 1996), pp. 268–285. ISSN: 0362-1340. DOI: 10.1145/236338.236363.

[25] Maximilian Störzer and Christian Koppen. “PCDiff: Attacking the Fragile Pointcut Problem, Abstract”. In: *European Interactive Workshop on Aspects in Software*. Berlin, Germany, Sept. 2004.

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