UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Reconciling Context-Oriented Programming and Feature Modeling

Dissertation presented by
Alexandre KÜHN

for obtaining the Master's degree in
Computer Science
Option: Software Engineering and Programming Systems

Supervisor
Kim MENS

Readers
Benoît DUHOUX, Xavier GILLARD

Academic year 2016-2017

Abstract

The advent of connected devices, such as smart-phones, enabled the conception of several of software applications that are yet more intelligent. At the moment, the information in these applications is mainly encoded by individuals, but we can imagine in the near future that sensors may produce and provide it without any human intervention.

With nowadays tools, it is very complicated to design smart software applications that are aware of their surrounding environment: programmers must handle all existing combinations of the contextual information at once, which increases the complexity of the program exponentially with the amount of information that is considered by these applications.

Some software engineering domains aim to ease the development of such applications, such as *Feature-Oriented Software Development* and *Context-Oriented Programming*. Although they have been studied in isolation and use different approaches, they seem to solve similar problems. Still, few work has been performed to reconcile ideas from both these domains.

The goal of this thesis is to reconcile these two domains, in order to ease the development of applications with highly dynamic behavioral changes. To do so, a new formalism is introduced, so as to design a framework to build these applications. The framework is tested with a self-assembled context-aware application, an *Emergency Response System* that helps users in case of emergency.

Acknowledgements

I would first like to thank my thesis promotor, Prof. Kim Mens, for his precious advices during the entire academic year. Each week, I took some of your time to share my problems with you on my thesis. Systematically, it took more time than what was planned, but I always left your office with so many solutions to my problems!

I would also like to thank my Assistant Lecturer, Benoît Duhoux, for all of his relevant remarks on my work. Your experience as a young Ph.D. student and not long ago former master student helped me to understand with precision how I could improve my work.

I would also like to acknowledge Xavier Gillard as the third reader of this thesis. I am grateful for your valuable comments on this thesis.

Finally, I must express my gratitude to my parents for providing me support and encouragement throughout my years of study, and through the process of writing this thesis. This accomplishment would not have been possible without you. Thank you.

Alexandre KÜHN

Contents

List of Figures

Listings

Acronyms

AOP	Aspect-Oriented Programming
CaaF	Context-as-a-Feature
COP	Context-Oriented Programming
CoPN	Context Petri-Net
DOP	Delta-Oriented Programming
DSL	Domain-Specific Language
ERS	Emergency Response System
FD	Feature Diagram
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
FOSD	Feature-Oriented Software Development
MPL	Multiple-Product-Line
OO	Object-Oriented
OOP	Object-Oriented Programming
SPL	Software Product Line

Chapter 1

Introduction

The advent of connected devices, such as smart-phones, enabled the conception of several of software applications that are yet more intelligent. As an example, Japanese citizens can get mobile applications that warn them in the presence of earthquakes or tsunamis in their surroundings. In particular, these applications alert the users when an earthquake of magnitude 4 or higher has been detected, or when a tsunami is about to come.

At the moment, the information in these applications is mainly encoded and provided by staff members of emergency agencies. In the near future, we can imagine that cities will become smart, so that this information will be produced by sensors spread across these cities, without any human intervention. These sensors may deduce the presence of earthquakes in the city, and send this information to many connected devices.

In addition to this information on the physical environment, the connected devices of the citizens may also provide information on their current state, so that the application could personalize even more its content. For instance, the application may provide real-time information to the user if his or her device is connected and has high battery power. On the other hand, it may restrain some features if the battery level is low.

With nowadays tools, it is very complicated to design this kind of applications at an industrial scale: the creator of these applications must create programs that are almost monolithic, where the program itself must handle all existing combinations of this information at once. As a consequence, these programs are usually large, not very modular, and poorly maintainable, given that the number of behavioral variations increases exponentially with the amount of information that are considered by these applications.

Some software engineering domains aim to build tools to ease the developement of such applications, such as FOSD and COP (see chapter 2). The resulting works from each of these domains are interesting, and some have already won the heart of some industrial software development

(e.g. SPLs). Although these two domains solve similar problems, they use different approaches. Few work has been carried out to reconcile the important results of both domains.

The goal of this thesis is to reconcile ideas from both engineering domains, in order to come up with a framework that will ease even more the development of smart systems that are aware of their surroundings.

With this goal in mind, we will start by retrieving important results from both domains, after which we will introduce a new formalism that brings both ideas together. Finally, based on this formalism, we will design a framework to create applications with highly dynamic behaviors, and we will validate our implementation by developing such an application.

This thesis brings the following contributions:

- It suggests a new formalism that reconciles ideas from two different software engineering domains, FOSD and COP;
- It provides an architecture and an implementation of a framework that rely on this new formalism;
- Furthermore, it tests a new modeling approach called CoPN (see Section 2.3.4).

First, we will summarize the important ideas from the two different engineering domains, FOSD and COP, in addition to briefly introduce the modeling approach CoPN (Chapter 2). The constituents of this chapter will let us define the approach that we followed in order to build our framework (Chapter 3). Then, we will present the application that we will use in order to validate our work (Chapter 4). After that, we will describe the architecture of the framework (Chapter 5), and we will explain its implementation mechanisms (Chapter 6). This chapter will also let us briefly discuss about the limitations and future work of the framework (Chapter 7), before we conclude on the accomplished work (Chapter 8).

The code of the framework is available on the following repository:

`https://github.com/alexkuhn/reconciling-cop-and-fm`

Chapter 2

Related Work

In this chapter, we present the knowledge and related work upon which our framework is based.

One of the core ideas of this thesis is the notion of *software feature*. We would like to design and implement software systems in terms of their features; it is therefore helpful to know more about this notion. There is a software engineering research domain that aims to build software systems from features, which is called ***Feature-Oriented Software Development*** (FOSD) [1]. We explain some of its results, because they have been used for the implementation of our framework.

We have implemented our framework on top of a programming paradigm called ***Context-Oriented Programming*** (COP) [17]. This paradigm lets us build a particular class of systems called "context-aware systems". We will therefore also explain some useful concepts that result from the research on this paradigm, starting from defining what is meant by the notion of "context".

The applications that run on top of our framework should be able to dynamically change their contexts and features. We will use a variant of the formalism of ***Petri Nets*** [26] to model the active states and dependency relationships of contexts and features. We explain the basics of Petri Nets, in addition to extensions that our particular variant of Petri Nets make use of [4].

2.1 Feature-Oriented Software Development

Feature-Oriented Software Development (FOSD) is a software engineering research domain that is dedicated to building software systems from building blocks called features. Our framework uses many results from FOSD, such as the modeling and the implementation techniques to build software systems from features.

2.1.1 Feature

The notion of feature is key to our framework, so it is important to clearly define what a feature is. We give some definitions, from least to most technical:

- *"A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"* [18];
- *"An increment of a program functionality"* [3];
- *"A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"* [2].

The last definition clearly says what is meant by software features programmatically: it is defined as several modifications or extensions that are applied to a given program code. That is exactly how features are defined in our framework.

2.1.2 Software Product Lines

FOSD relies on many tools and techniques to build software systems called *Software Product Lines* (SPLs). In fact, SPLs have first been studied outside of FOSD, which is why they use the term "software assets" instead of "software features". Basically, instead of building a single system as the composition of software assets, SPLs builds a family of systems from several software assets [34]. A family of systems is a group of systems that share some of these assets. Assets that are shared between all of these systems are called *commonalities*. Assets that are partially shared or unique to some of these systems are called *variabilities*.

The process of SPLs is as followed (Figure 2.1): from a set of software assets (i.e. features in FOSD) and a set of models, a configurator mixes them in order to produce software systems. Each model defines the list of all selected software assets that compose one of these software system. These models are called *Variability Models* because they highlight the commonalities and variabilities of these software products [28].

2.1.3 Feature Modeling

Feature Modeling is a way of defining Variability Models, where assets are features. It originates from a technical report written by Kang et al. [18] in which they present a modeling approach to build SPLs with the notion of feature (*Feature-Oriented Domain Analysis*, FODA for short). In this method, the features of a given family of software systems are modeled

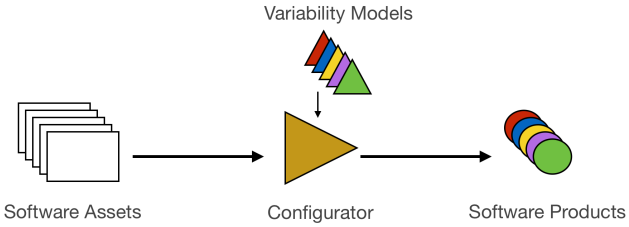


Figure 2.1 – Software Product Lines in a Nutshell [21].

in a particular way called *Feature Diagram*.

A **Feature Diagram** is a tree, where nodes are features, and edges are constraint relations between a particular feature and its parent feature. For instance, figure 2.2 shows a simplified feature diagram of E-Shop services. The root of the tree is named with the considered family of software systems, in this case `EShop`. We consider that all E-Shops must necessarily have the features `Catalogue`, `Payment` and `Security`. The feature `Search` is optional, meaning that only some E-Shops have this feature. If the system possesses the feature `Payment`, it must possess either `BankTransfer` or `CreditCard`, or both of them. If the system possesses the feature `Security`, it must possess either `High` or `Standard`, but not both of them. Finally, if the system possesses the feature `CreditCard`, it must also possess the feature `High`.

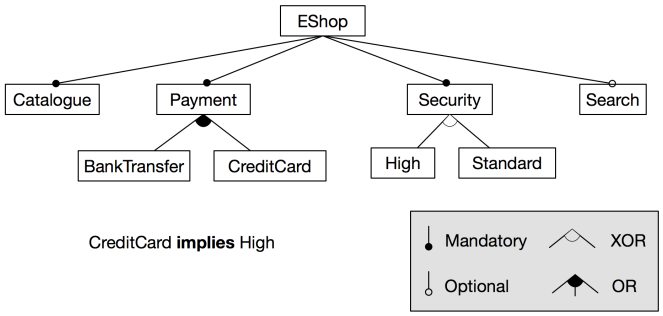


Figure 2.2 – Feature Diagram for E-Shop Services [1].

A selection of features is called a *configuration*. A configuration is said to be *valid* if it obeys the constraints of the feature model, otherwise it is said to be *invalid*. Features cannot be selected without their parents.

Nowadays, Feature Diagrams have become the standard way of modeling features in FOSD. This modeling notation lets us define coarse-grained features as the composition of finer-grained features. For instance, the feature `Payment` is composed of either `BankTransfer` or `CreditCard`,

[1] A feature diagram representing a configurable e-shop system. `https://en.wikipedia.org/wiki/Feature_model#/media/File:E-shopFM.jpg`. Accessed: 2017-08-18.

or both of them.

Hartmann et al. proposed an interesting extension to Feature Diagram called ***MPL-Feature Model*** (MPL stands for *Multiple-Product-Lines*, illustrated in Figure 2.3) [15]. This model contains two sub-trees, the *Feature Model* on the right, and the *Context Variability Model* on the left. A Context Variability Model is a model that organizes something called *context* in a hierarchy, similarly to features in the Feature Diagram (we define what a context is later in the chapter (see Section 2.2.1), but you don't have to know it for the moment). Both sub-trees are connected together by means of dependencies between contexts and features. Hartmann et al. explain that this new model enables us to build multiple software product lines based on some criteria, here called contexts. That way, we can put constraints on the selection of the features, based on the selection of the contexts. For example, if we model Car Navigation Systems, the selection of the context `Belgium` may require the selection of the feature `BelgianNavigationMap`.

MPL-Feature Models are a possible way of reconciling contexts and features from the perspective of FOSD.

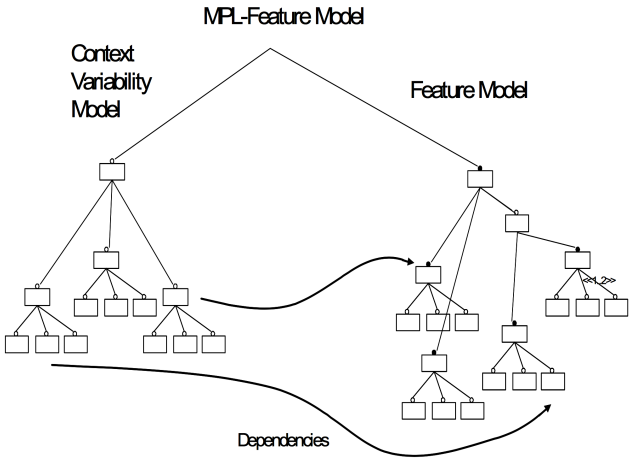


Figure 2.3 – MPL-Feature Model [15].

2.1.4 Feature Interaction Problem

The composition of fine-grained features into a coarser-grained feature is not as trivial as it appears. A naive way is to pile the codes of all fine-grained features like an UNION operation in order to form the resulting coarser-grained feature. This approach creates an issue called the **Feature Interaction Problem** [30].

We illustrate this problem with the following scenario (Figure 2.4): suppose that we have two distinct non-circular singly-linked list implementations `SINGLE` and `REVERSE` that we want to compose in order to

have a non-circular doubly-linked list `SINGLEREVERSE`. Each list has its own attributes and methods.

The methods of each list (a) and (b) obey the specifications of the lists, which is to update their attributes so that they refer to the first or the last node of the list. By simply merging the codes, using a technique called *Superimposition* [1], we get an incorrect implementation of the doubly-linked list `SINGLEREVERSE` (c), because the methods no longer obey the specifications of the list. To prove it, suppose that the list is initially empty (the attributes `first` and `last` are assigned to *null*). If we *push* a node `A`, the attribute `first` will correctly point to `A`, but the attribute `last` will still point to *null* instead of `A`. We will have a similar problem with the method `shup`, and this issue is due to missing instructions in both methods (d).

```
1 class List {
2   Node first;
3   void push(Node n) {
4     n.next = first; first = n;
5   }
6 }
7 class Node {
8   Node next;
9 }
```

(a) SINGLE

```
1 class List {
2   Node last;
3   void shup(Node n) {
4     n.prev = last; last = n;
5   }
6 }
7 class Node {
8   Node prev;
9 }
```

(b) REVERSE

```
1  class List {
2    Node first; Node last;
3    void push(Node n) {
4      n.next = first; first = n;
5    }
6    void shup(Node n) {
7      n.prev = last; last = n;
8    }
9  }
10 class Node {
11   Node next; Node prev;
12 }
```

(c) wrong SINGLEREVERSE

```
1  class List {
2    Node first; Node last;
3    void push(Node n) {
4      if(first==null) last=n; else first.prev = n;
5      n.next = first; first = n;
6    }
7    void shup(Node n) {
8      if(last==null) first=n; else last.next = n;
9      n.prev = last; last = n;
10   }
11 }
12 class Node {
13   Node next; Node prev;
14 }
```

(d) correct SINGLEREVERSE

Figure 2.4 – Feature Composition with Interaction Problem [1].

This small example illustrates the Feature Interaction Problem, which forces us to explicitly specify the changes to the codes when many features interact together. Some researchers proposed some solutions to this problem, such as the notion of *Derivatives* [22], which are pieces of code that explicitly specify how to refine features when there are interactions.

2.1.5 Feature Implementation

Several implementation techniques can be used for building SPLs [33]:

- **Preprocessor directives**, which are precompiled conditionals to set variability points in the code, such as the `#ifdef` annotation in C.
- **Aspect-Oriented Programming** (AOP) [20], which lets us implement a special class of features called concerns in a modular way. These features extend or modify a base program in many places of the code, so their logic is scattered over the resulting code.
- **Feature-Oriented Programming** (FOP) [30], which is an extension of OOP. Software systems are built by composing feature modules, which are themselves composed of classes. *rbFeatures* [14] is a FOP implementation on top of the RUBY programming language.
- **Delta-Oriented Programming** (DOP) [32], which implements features as delta modules. These modules specify the changes to apply on the base program, in order to add, modify, or remove some code.

Most programming languages support the first technique, but it becomes hard to maintain the code because of all the conditional branches it generates. The last 3 techniques have inspired the way features are implemented in our framework, in addition to a fifth technique, called *Context-Oriented Programming* (COP). We explain COP in the next section.

2.2 Context-Oriented Programming

Context-Oriented Programming (COP) is a programming language paradigm to build software applications that can vary their behavior depending on changing contexts, in a modular way [17]. We mainly use this programming paradigm to implement our framework.

2.2.1 Context

COP provides tools to build software systems with dynamic behavioral variations that depend of their surrounding context. Let's first define what is meant by "context", similarly to the previous section with the meaning of "feature". Some researchers define this concept as follow:

- *"Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."* [9].

– *"Any information that may have impact on the behaviour of the program."* [19]

The first definition is the commonly accepted one by the scientific community. The second definition is an oversimplified version of the former one. Nevertheless, it gives a good technical and intuitive understanding of what is a context. Suppose that we have a program that varies its behavior by means of conditional statements in its code. The conditions can be seen as "contexts". A context is either *active* or *inactive*, similarly to a condition being either true or false.

Contexts are split in two categories: *external* and *internal*. For instance, the localization `Belgium` is an external context, while the battery status `LowBattery` is an internal context. The set of all external contexts is called the *physical environment*, while the set of all internal contexts is called the *logical environment*. Contexts that relate to user's interactions are usually considered as internal (e.g. `ColourBlind`, a context that has been selected by the user).

2.2.2 Context-Aware Systems

A software system is said to be "context-aware if it can extract, interpret and use context information and adapt its functionality to the current context of use" [31]. It is a class of software systems that *"uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task"* [9]. These systems become more and more common today, mainly due to the rise of technically advanced applications. For instance, smartphones are pocket computers containing a lot of applications with context-dependent behaviors. In particular, these applications make use of the device's sensors. To name a few of these behavioral variations from sensors:

– The gyroscope detects the orientation of the device, which can be used to optimize the display of some content.

– The GPS chip detects the location of the device, which can be used to suggest close restaurants.

– The front-facing camera detects the surrounding light, which can be used to automatically adjust the brightness level of the display screen.

An easy way to implement context-aware systems is to make use of conditional statements. For instance, when a phone receives a call, its behavior varies depending on the ring mode that has been selected by the user (Listing 2.1). If the selected ring mode is `quiet`, then the phone vibrates. If the selected ring mode is `busy`, then the phone forwards the call. The default behavior is to play the ringtone.

```
class Phone
   def receive(call)
      if quiet then phone.vibrate
      elsif busy then phone.forward_call
      else phone.play_ringtone
   end
end
```

Listing 2.1 – Receive a call depending on the ring mode, with conditionals [23].

It is not a good solution to design context-aware applications with conditionals, because the code becomes very complicated and its logic is tangled and scattered [23]. Delegation in Object Oriented Programming is not an acceptable alternative, because it produces undesirable effects (e.g. *Object Schizophrenia*[2] [16]). Furthermore, both of these solutions lack "dynamic flexibility", as they do not let applications change their behavior at run-time on situations that were not anticipated during the development process [8].

Here comes a new solution into play, namely **Context-Oriented Programming** (COP), which is a programming paradigm that provides tools and concepts to develop context-dependent behavioral systems. To do so, COP offers a general architecture to build context-aware systems (Figure 2.5):

- The module `Context Discovery` retrieves data from sensors that capture the environment ("world" in the schema), in order to detect some changes to the active state of some external contexts for the module `Context Management`.
- The `Application Behaviour` varies depending on the active contexts. The application provides some information and services to the user (physical world), and can also detect changes to the active state of some internal contexts for the module `Context Management`.
- The module `Context Management` receives some requests to change the active state of some internal and external contexts, and chooses the context changes to apply, in order to get the new coherent set of `Active Context`s.
- The `Active Context`s dictate the behavior of the application.

The module `Context Management` makes its decision based on the detected contexts by the `Context Discovery` and the `Application Behaviour`,

[2] A situation that *"results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identify)"*.

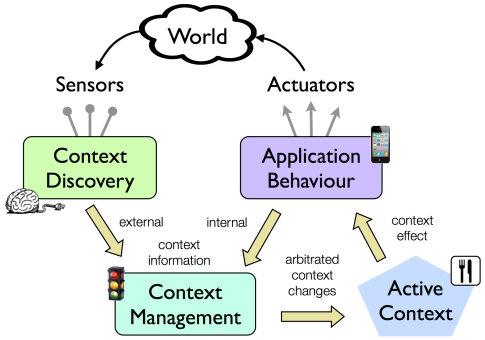


Figure 2.5 – General COP Architecture for Context-Aware Systems [12].

in addition to the relations between the contexts which are called **Dependencies**. These relations declare constraining (and possibly cascading) dependencies between contexts. For instance, a request to activate the context C_A may be refused due to the context C_B being active, or a request to activate the context C_C may cascade a request to activate the context C_D.

A recent COP architecture has been proposed by Mens et al. [10, 24] (Figure 2.6). This is a layered architecture that is organized as followed:

Interaction This layer gathers the data from the external and internal environments, that is from the user and from the sensors. The sensors capture data from the physical world (e.g., the surrounding noise in dB) and the computing platform (e.g., the battery level of the device in %).

Discovery This layer interprets and reasons on the collected data, in order to request the activation or deactivation of some contexts.

Handling This layer determines the contexts to activate or to deactivate, then selects the features to activate or to deactivate, and finally executes the active features.

Application This layer contains all bits of the architecture that are application-specific. In other words, the developer of the application can configure some parts of the previous layers in this layer, such as the declaration of the contexts and their dependency relations.

This COP architecture extends the general architecture (Figure 2.5) with the notion of *feature*. It has been inspired by the MPL-Feature Model from Hartmann et al. (see Section 2.1.3) and prototypical COP architectures already incorporating features [5, 13]. It reconciles contexts and features from the perspective of COP.

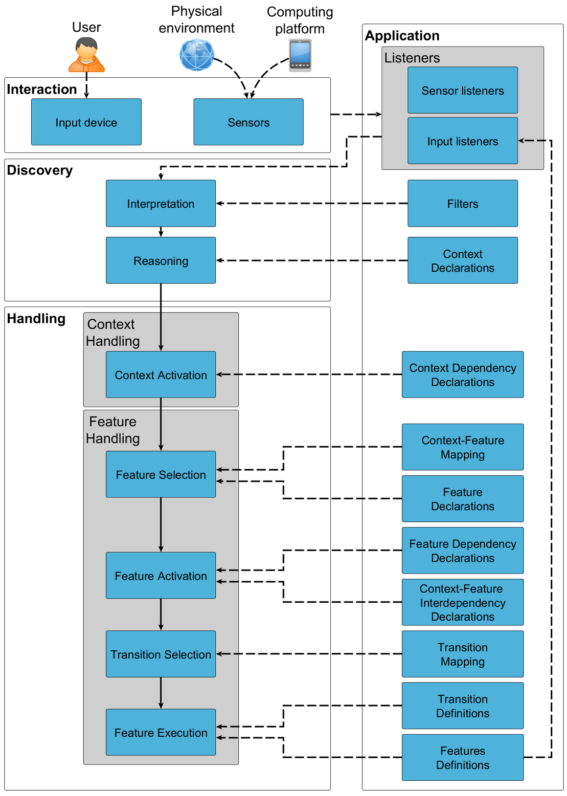


Figure 2.6 – Layered COP Architecture for Context-Aware Systems [10].

2.2.3 Context Dependencies

There are many types of dependency relations between contexts. We briefly explain some of them coming from González et al. [12], but we use the updated terminology from Cardozo et al. [6]. All examples (and some pictures) are coming from González et al. [12].

Exclusion the exclusion relation refuses the activation of a context if another context is already active. This link is useful when we want to have contexts that have no reason to be active simultaneously. For instance, if we have the contexts `LowBattery` and `HighBattery`, we do not want to have both contexts being active at the same time (Figure 2.7).

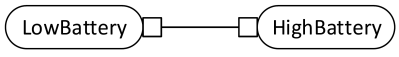


Figure 2.7 – Exclusion between `LowBattery` and `HighBattery`.

Requirement the requirement relation authorizes the activation of a context only if another context is active, otherwise it is refused. This relation is useful when we want to have a context that has

no reason to become active if another context is not already active. For instance, if we have the contexts `HDVideo` and `HighBattery`, we would like to authorize the activation of `HDVideo` only if `HighBattery` is active (Figure 2.8).

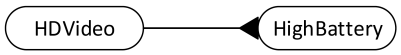


Figure 2.8 – Requirement between `HDVideo` and `HighBattery`.

Implication the implication relation allows us to have a cascading link between two contexts, so that one of these contexts "is included in" the other one. For instance, if we have the contexts `Brussels` and `Belgium`, and a GPS chip detects that we are in `Brussels`, then we are also in `Belgium`. However, if the sensor detects that we are no longer in `Belgium`, then we are inevitably not in `Brussels` anymore (Figure 2.9).

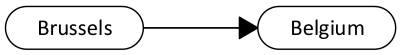


Figure 2.9 – Implication between `Brussels` and `Belgium`.

Causality the causality relation lets us have a cascading link between two contexts, but this one is weaker than the implication relation. This relation is useful when we want to have a one-way cascading relationship between contexts. For instance, if we have the contexts `Cafeteria` and `Noisy`, we know that most cafeterias are quite noisy, so the activation of `Cafeteria` could activate `Noisy` by default. However, there are some cafeterias that are not noisy, therefore it is allowed to deactivate `Noisy` while keeping `Cafeteria` active (contrary to the relation between `Brussels` and `Belgium`) (Figure 2.10).

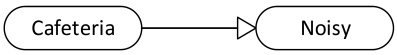


Figure 2.10 – Causality between `Cafeteria` and `Noisy`.

As a consequence of some of these dependencies, the COP system must deal with *multiple activations of a same context* in order to be consistent [6]. To illustrate why this system must support multiple activations, suppose that we have a context `Connectivity` that can be activated by the activation of any one of these contexts, `WLAN`, `WiFi` and `3G`, by means of implication relations (Figure 2.11). If all of these contexts are active, and we deactivate `3G`, we want to keep `Connectivity` active as long as `WLAN` or `WiFi` are active. In our framework, we model the state of the contexts with *Context Petri Nets* (CoPN, see Section

2.3.4), and this model supports multiple activations. Intuitively, a context with multiple activations will be modeled as a place in the Petri net that contains multiple tokens (see Section 2.3).

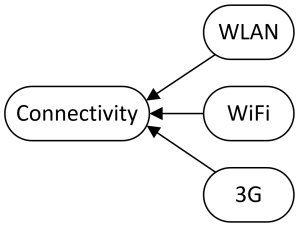


Figure 2.11 – Multiple Activation of `Connectivity`.

2.2.4 Adaptation

As explained before, when we described the COP architectures to build context-aware systems, we said that the behavior of the application is affected by the changing contexts. Each context possesses a set of contextualized behaviors, which are commonly called **Adaptations** (or *Layers* in some flavors of context-oriented programming [35]).

An adaptation is a context-specific logic associated to a procedure. Since most COP implementations are built on top of an OOP language, adaptations are usually associated with a class and a selector (i.e., a method name and its arguments). Listing 2.2 shows how adaptations are defined in PHENOMENAL GEM [29], a COP framework on top of RUBY ON RAILS, using the example of the Phone call from Listing 2.1: the context `quiet` contains an adaptation for the class `Phone` and the selector `call`, so that when the context `quiet` is active and the phone receives a call, it vibrates.

```
context :quiet do
   adaptations_for :Phone
   adapt :call do
      phone.vibrate
   end
end
```

Listing 2.2 – Adaptation on class `Phone` and selector `call` for the context `quiet` in PHENOMENAL GEM [29].

As we said before, when contexts change, the behavior of the application must be adapted depending on the newly active or inactive contexts. The most common technique to do so is called **Method Pre-Dispatch** [12], and it works as followed:

– When a context is activated or deactivated, the module *Context Management* finds all active adaptations for every class and selector this context adapts to.

– Then, it sorts out them according to a composition policy, from the most-specific to the least-specific one (e.g. based on the activation age of their corresponding context).

– Finally, it deploys the first adaptation (i.e. the most specific one), so that any class that receives a message on one of their adapted selectors will have the logic of the most specific adaptation, according to the chosen policy.

The main inconvenience of this technique is to have a non-negligible overhead, because the system must compute the sorted lists of adaptations each time a context is activated or deactivated. Its algorithm is illustrated in listing 2.3.

```
when a context c is (de)activated do
    for each class and selector c adapts do
        Find all active methods
            M(c, s) = {m_1, m_2, m_3, ..., m_n}
        Reorder then according to policy
            m_1 < m_2 < m_3 < ... < m_n
        Deploy the first one
            m_1
    end
end
```

Listing 2.3 – Method Pre-Dispatch Algorithm [23].

An adaptation m_i can call the next adaptation m_{i+1} in the list by using the keyword `proceed`, thus the most-specific adaptation can simply reuse codes from less-specific adaptations. It works in a similar way as the keyword `super` in most OOP languages, where the adaptation m_i is like a class that "inherits" from another class, here m_{i+1}. A big difference, however, is that these adaptations are connected dynamically, unlike the super-class relations which are declared statically. Another advantage of the keyword `proceed` is to have adaptations that simply *refine* the current behavior of an application, and we can stack refinements like the well-known Decorator Design Pattern in OOP.

For instance, if we want to refine a phone with *call screening*, we can add a context `call_screening` containing an adaptation on the class `Phone` and the method `call`. This adaptation uses the `with_screening` feature on top of the other contextualized behavior (*vibrate*, *forward_ call*, or *play_ ringtone*, depending on the active contexts). Listing 2.4) contains the declaration of the context `call_screening` in PHENOMENAL GEM.

```
context :call_screening do
   adaptations_for :Phone
   adapt :call do
      phone.with_screening { proceed }
   end
end
```

Listing 2.4 – Adaptation on class `Phone` and selector `call` for the context `call_screening` in PHENOMENAL GEM [29].

2.2.5 Context Interaction Problem

Similarly to the Feature Interaction Problem in FOSD, there can be conflicts between adaptations. These conflicts can be due to the adaptations themselves, the contexts, or even the sensors (e.g. defective sensors). This problem is called the **Context Interaction Problem** [25].

Let's consider, for example, a home automation system with an emergency response module that reacts to detected emergency situations. If the house is burning, the system detects it and activates the sprinklers. If the house has water damages, the system detects it and turns off the water supply. What if the house has water damage and is burning at the same time? The system may activate the sprinklers, but since the water supply has been deactivated, these sprinklers won't stop the flames, and the system will let the house collapse from burning. We might have preferred to stop the flames, at the cost of aggravating the water damage problem.

Method Pre-Dispatch is not sufficient to solve the context interaction problem, because it only selects the most appropriate adaptations *after* activating the contexts. The conflict might be detected *before* that stage, such as when the dependency relations are checked. It is therefore essential to have strategies to handle such conflicts, at the moment they are detected.

In our framework, we use strategies to solve the context interaction problem at the level of contexts, and the detection of such conflicts is made possible by an extension of Petri Nets, which is explained in the next section below.

2.3 Petri Nets

Petri Nets [26] are mathematical and graphical modeling methods that are applicable to a very wide variety of applications. We use an extension of Petri Nets in our framework, in order to model the state of contexts

and features, as well as their inter-dependencies.

2.3.1 Basics

A **Petri Net** is a directed, weighted and bipartite graph N containing two types of nodes: *places* (noted p, graphically as a circle) and *transitions* (noted t, graphically as a rectangle). The arcs are directed, either from a place to a transition ($p \rightarrow t$) or from a transition to a place ($t \rightarrow p$). An arc is labeled with its weight w, which is a strictly positive integer. No label means a weight of 1. The input (respectively, *output*) places of a transition t are denoted by $\bullet t$ (respect., *by* $t\bullet$).

A transition t can be interpreted as an event, its input places $\bullet t$ can be seen as the pre-conditions, and its output places $t\bullet$ can be seen as the post-conditions. The state of the Petri Net is changed based on the following ***firing rules***:

1. A transition t is said to be *enabled* if each place p in $\bullet t$ has at least $w(p, t)$ tokens.
2. An enabled transition may or may not fire (it fires if the event occurs).
3. The firing of an enabled transition removes $w(p, t)$ tokens from each place p in $\bullet t$, and adds $w(t, p)$ tokens to each place p in $t\bullet$.

Figure 2.12 illustrates the firing rules with 3 places and a single transition, with the chemical reaction $2H_2 + O_2 \longrightarrow 2H_2O$ [26]. Initially, we have 2 units of H_2 and O_2, which are modeled with 2 places. The product of the chemical reaction H_2O is also modeled as a place. The transition t connects all the places, so that the transition is enabled if and only if there are at least 2 tokens in H_2 and at least one token in O_2. When t fires, it adds 2 tokens to H_2O.

Many extensions have been proposed to Petri Nets, leading to many new applications. Context Petri Nets (CoPN) [6] extends basics Petri Nets in order to model relationships between contexts and features in the proposed framework implementation of this thesis. CoPN mainly relies on two smaller extensions of Petri Nets, *Inhibitor Arcs* [26] and *Reactive Petri Nets* [11]. Both of them will be explained before CoPN.

2.3.2 Inhibitor Arcs

Inhibitor arcs are a new type of arcs directed from a place p to a transition t ($p \multimap t$). $\bullet t$ denotes the input places of t from normal arcs, while $\circ t$ denotes the input places of t from inhibitor arcs. It enhances the first rule of the *firing rules*: a transition t is said to be *enabled* if each place

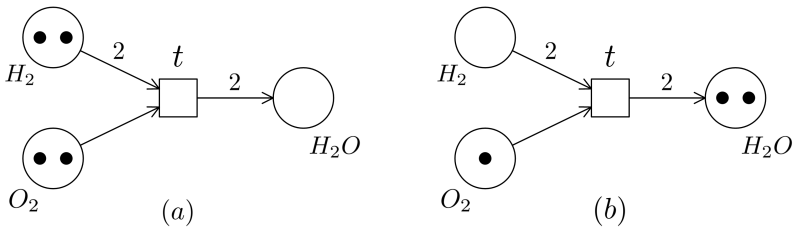


Figure 2.12 – Example of firing an enabled transition, (a) Before firing t, (b) After firing t [26].

p_i in $\bullet t$ has at least $w(p_i, t)$ tokens and if each place p_j in $\circ t$ has no token.

Figure 2.13 shows an example of a transition that is not enabled due to its single inhibitor arc: the transition is enabled if and only if the place p_1 contains a token and the place p_2 contains *no* token (which is not the case).

Some behaviors cannot be modeled without using inhibitor arcs: they add the notion of "test zero", which puts this kind of Petri Nets at the same level of effectiveness as Turing machines [27].

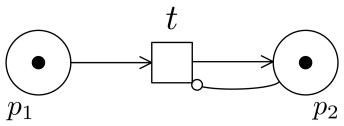


Figure 2.13 – Our example with inhibitor arcs, preventing the transition t from being enabled.

2.3.3 Reactive Petri Nets

In the non-reactive version of Petri Nets, an enabled transition may or may not fire, so the state of the Petri Net does not influence the firing of transitions. Reactive Petri Nets [11] remedy to this problem, by enabling the modeling of the behavior of reactive systems. To do so, it splits transitions into two categories: *external* transitions (white boxes), which are equivalent to the original transitions, and *internal* transitions (black boxes), which must fire when they are enabled.

Figure 2.14 shows a Reactive Petri Net, where initially the place p_1 has 1 token (a). The external transition t_{ext} is enabled, and when it fires, the place p_1 contains no token, and the place p_2 contains 1 token. Since the internal transition t_{int} is enabled, it fires, which result of the place p_1 having 2 tokens, while place p_2 contains no token. If many internal transitions become enabled at the same time, there is a free choice from the system.

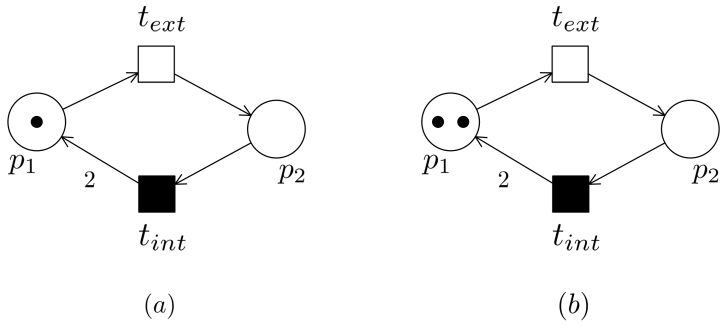


Figure 2.14 – Our example with Reactive Petri Nets (a) before firing t_{ext}, (b) After firing t_{ext}.

2.3.4 Context Petri Nets

Context Petri Nets (CoPN) make use of both extensions in order to model the active state of contexts, as well as the dependency relations between contexts. In addition to the above smaller extensions, CoPN split places into two types: *context places*, which represent the state of the contexts (graphically, solid circles), and *temporary places*, which represent the preparation stage for the activation or deactivation of the contexts (graphically, dashed circles). Figure 2.15 illustrates how a single context (a.k.a., a singleton) is modeled with CoPN.

The temporary places are useful to check the consistency of the system. A system is said to be inconsistent if there are tokens in some temporary places while no internal transition is enabled. This situation occurs when some contexts couldn't be activated or deactivated, so they show conflicts during the activation of contexts (see Section 2.2.5). In our framework, temporary places let us detect these conflicts, so that we can have a strategy to handle them.

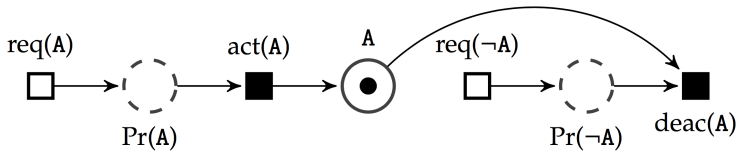
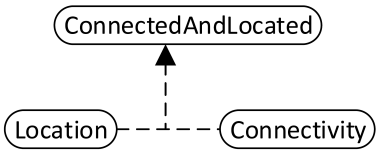


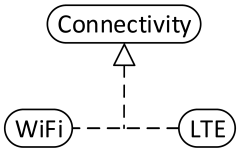
Figure 2.15 – Context A modeled in CoPN [6].

CoPN models dependencies between contexts by adding arcs and internal transitions in the Petri Net. These new elements link multiple singletons together, in order to have cascading and constraining properties among them. CoPN supports many dependency relations, such as the one that we have introduced in Section 2.2.3. In addition to these relations, CoPN adds the dependencies conjunction and disjunction, whose graphical representations are illustrated in Figure 2.16.

We show how these dependency relations are modeled in CoPN with the requirement relation from Figure 2.17, where the context `HDVideo`



(a) Conjunctive context `ConnectedAndLocated` between `Location` and `Connectivity`.



(b) Disjunctive Context `Connectivity` between `WiFi` and `LTE`.

Figure 2.16 – Conjunctive and Disjunction Relations in CoPN.

requires the context `HighBattery`. It enhances the modeling of the singletons by adding arcs and internal transitions, in **blue** and in **magenta** in the figure. The two colors denote the two underlying functions that is defined inside a dependency relation in CoPN: an ***extending*** and a ***constraining*** interaction between the considered contexts.

The reason of distinguishing these two functions is that constraints depend on the extensions. In order to apply multiple dependencies on the model, we must proceed in two steps: first, we apply the extending interactions of all these dependencies; second, we apply their constraining interactions. If we don't apply the constraints after the extensions, we might miss some important constraining relations [4].

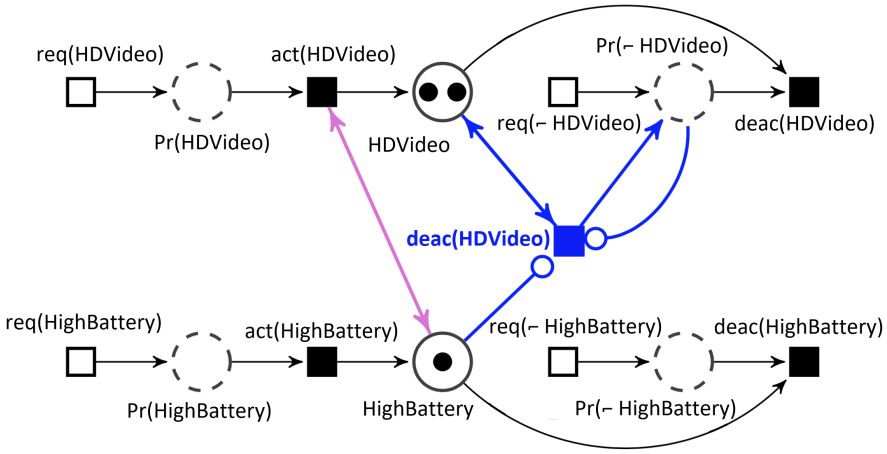


Figure 2.17 – The Requirement Dependency Relation Between `HDVideo` and `HighBattery` (`HDVideo` requires `HighBattery`), adapted figure from [6].

Chapter 3

Approach

In this chapter, we describe the approach that lets us build our solution, using the elements described in the previous chapter.

First, we motivate the need for our solution. Second, we introduce and justify the programming language that we use to implement it. Third, we explain and justify some design choices.

We show in later chapters the architecture and the implementation of our framework, which both heavily rely on our design choices that are presented in this chapter (see Section 5 and 6).

3.1 Motivation

FOSD and COP are two engineering domains that aim to achieve ***Software variability*** [7]: FOSD provides functionality by composing features, and COP brings it with run-time behavior modifications. They do seem to be converging, as we observe many coinciding ideas:

- They both work on a family of systems of particular domains: FOSD builds *Software Product Lines*, while COP builds *Context-Aware Systems* (see Sections 2.1.2 and 2.2.2).
- They have strong similarities in some of their implementation techniques: *deltas* in DOP, similarly to *adaptations* in COP, are fine-grained behavioral changes that are applied to a base program (see Sections 2.1.5 and 2.2.4)
- Both domains come accross similar problems: FOSD shows the *Feature Interaction Problem*, while COP shows the *Context Interaction Problem* (see Sections 2.1.4 and 2.2.5)
- Each of them expands by incorporating the key concept of the other domain: FOSD expands with *contexts* in the MPL-Feature Model, and COP expands with *features* in the Layered COP Architecture (see Sections 2.1.3 and 2.2.2)

There are however still significant differences between these two engineering domains:

- FOSD traditionally builds software systems at *configuration-time*, while COP always builds them at *run-time* [7].
- FOSD connects features *hierarchically* with Feature Diagrams, while COP connects contexts through *dependency* relations (see Sections 2.1.3 and 2.2.3).

We propose a COP language to build context-aware applications, so that developers of these applications can make changes to contexts and features at any time. It mainly differs from traditional COP languages in two aspects: it incorporates features as first-class entities, and it models contexts and features with dependency and hierarchical relations.

Before we list these design choices, we introduce and justify the programming language that we have used to implement our framework.

3.2 Programming Language

We choose the Ruby programming language to implement our solution. This choice is explained by former achievements of implementing COP mechanisms with this programming language [24, 29].

Ruby is an object-oriented scripted programming language that has been inspired by Perl for its syntax and Smalltalk for its semantics. It has many advanced reflective facilities, which is the main reason why this programming language is a good candidate for implementing COP mechanisms: we would like to make changes to the program's code at runtime (*intercession*), which is something that many existing programming languages lack or have limited support for (e.g. Java).

3.3 Design Choices

We have made several design choices in order to implement our solution, a framework for context-aware applications that uses contexts and features. Some of them introduce new concepts, which are defined in the subsections below.

3.3.1 Activatable

We group similarities between contexts and features in a new concept called `Activatable`. The main properties of contexts and features are that they have a *name*, and can be either *active* or *inactive*.

Contexts and features are kept as two separated entities, so that none of them is a special case of the other one. It is unlike different approaches, such as in Phenomenal Gem [29] where a context is considered as a particular kind of feature (*Context-as-a-Feature*, CaaF).

3.3.2 Alteration

Contexts and features differ mainly by what a feature additionally brings to the table: it provides pieces of code that *alter* a base program, so that the behavior of the resulting program shows that it possesses this feature. Theses pieces of code are called `Alterations`.

Since our language works on top of the OO paradigm, alterations *add*, *modify*, or *remove* data or functions of some objects or classes in a given program. Alterations can be seen as *deltas* in DOP. They can also be seen as *adaptations* in COP, which are extended to data.

3.3.3 Modeling

As we saw in the previous chapter, features are usually modeled by means of a *Feature Diagram* in FOSD, while contexts are usually modeled by means of *dependency relations* in COP.

Figure 3.1 summarizes the design choices for the modeling of contexts and features.

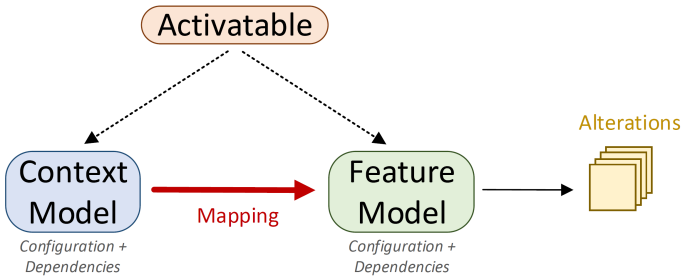


Figure 3.1 – Summary of the Modeling.

We separate contexts and features in their dedicated models, namely the ***Context Model*** and the ***Feature Model***. A ***Mapping*** defines which active context *enables* or *disables* a particular feature. Each of these models connects their entities with relations. There are two types of relation: *configuration* and *dependency* relations. Active features in the Feature Model determine the alterations that are applied on the base program.

Mapping

A Mapping links a context and a feature together: it sets a condition on the activation or the deactivation of the feature, based on the given context. The context is called a *mapped context*, and the feature is called a *mapped feature*. There are two types of mapping:

- *Disablers*: when the mapped context is active and the mapped feature is active, the mapped feature is deactivated.
- *Enablers*: when the mapped context is active and the mapped feature is inactive, the mapped feature *may* be activated: the feature is actually activated if none of its disablers has its conditions met.

Formally, if f is a feature, e_i are the conditions of its enablers ($\forall i = 1 \ldots n$), and d_j are the conditions of its disablers ($\forall j = 1 \ldots m$):

Activate f	*if*	$(e_1 \vee e_2 \ldots \vee e_i \vee \ldots e_n)$ $\wedge \neg(d_1 \vee d_2 \ldots \vee d_j \vee \ldots d_m)$ $\wedge$ f is inactive.	(1)
Deactivate f	*if*	$(d_1 \vee d_2 \ldots \vee d_j \vee \ldots d_m)$ $\wedge$ f is active;	(2)
	if	$\neg(e_1 \vee d_2 \ldots \vee e_i \vee \ldots e_n)$ $\wedge \neg(d_1 \vee d_2 \ldots \vee d_j \vee \ldots d_m)$ $\wedge$ f is active.	(3)

The condition (3) handles the case where the feature was active because of one of its enabling context: if the context becomes inactive, we must deactivate the feature. Figure 3.2 shows concretely how a feature f is activated or deactivated by its enabling context e and its disabling context d. The numbers in bracket below the blue arrows indicates the condition to deduce whether we should activate or deactivate f.

Configuration and Dependency Models

The Context Model and the Feature Model both have sub-models to model their inter-relations:

- ***Configuration Model***, which says when a set of active and inactive activatable entities, i.e. a configuration, is valid or not. It is a generalization of the *Feature Diagram* on any activatable entity.
- ***Dependency Model***, which defines the dependency relations between activatable entities. It is a generalization of the dependency relations defined in *Context Petri-Nets* on any activatable entity.

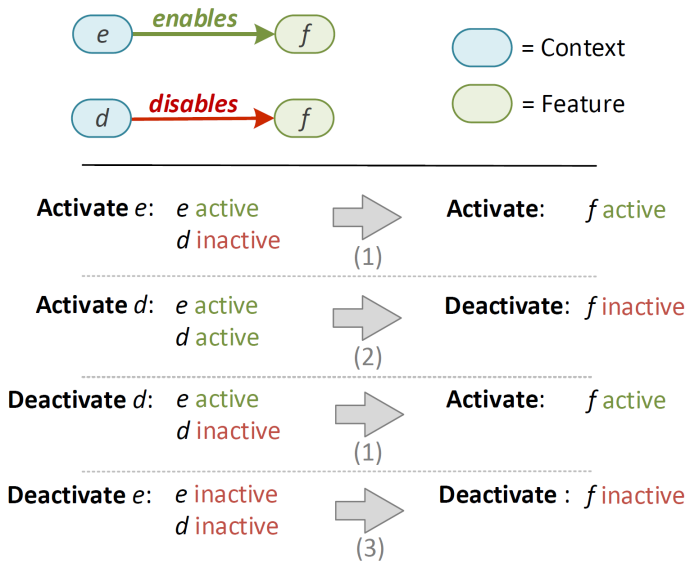


Figure 3.2 – Example of Mapping.

Conflict Resolution Policy

It is possible that the activation or deactivation of an activatable entity creates conflicts: some entities might be active (or inactive) although they shouldn't be. Conflicts are detected by the Context Model or the Feature Model, using the configuration and dependency sub-models.

There are many ways to resolve these conflicts, as stated in the *Context Interaction Problem* (see Section 2.2.5). For instance, we may use these usual conflict resolution strategies:

- *Ignore:* when a conflict is detected in the Context or Feature Model, it continues as if there was no conflict at all. This is equivalent to not having any strategy at all.
- *Rollback:* when a conflict is detected in the Context or Feature Model, it reverts the state of the current model to the most recent stable state. A model is said to be stable when it has no conflict.
- *Attempt:* when a conflict is detected in the Context or Feature Model, it attempts to resolve it by activating or deactivating entities that may be the cause of the conflict.

In our framework, we have implemented a compound strategy that uses an *Attempt* strategy whenever it can, otherwise it uses *Rollback*.

Chapter 4

Case Study

In this chapter, we define the case study used to develop a context-aware application with our framework.

First, we briefly explain what an *Emergency Response System* (ERS) is, the domain of systems in which our application is included. Secondly, we state the scope of functionality of our application. Thirdly, we show the model of the application in terms of contexts, features, and their corresponding inter-dependencies. Fourthly, we present the scenario that we use to test our application, illustrated by some screen-shots of the application.

4.1 Emergency Response System

Our case study focuses on a certain class of systems called ***Emergency Response Systems*** (ERS, for short). The main concern of these systems is to help saving people involved in emergencies. For example, if a wildfire is detected, an ERS can inform firefighters. These emergencies can be detected upon receipt of emergency calls, or from reasoning coming from sensors. Some cars are embedded with ERS, so that when an accident is detected, emergency services are automatically notified.

ERS are not simply limited to the notification of emergency teams: they can also inform users of an ERS application about emergencies. In particular, they can alert users that are in danger, and consequently guide them to safety. At least, they can provide general instructions whenever their users are facing an emergency, e.g. by suggesting them to stay calm and by advising them to check their own safety before helping someone else.

4.2 Functionality

Our ERS application is restricted to the user-side, and is mainly based on showing information to the user. We assume the data are provided by an external application (e.g. the administrative-side of the ERS).

The resulting ERS application possesses the 3 following functional aspects:

- **Alert:** it alerts the user when he is in danger, which is the case when he is inside or nearby a dangerous area.
- **Inform:** it informs the user about surrounding emergencies, dangerous zones and available safe places.
- **Guide:** it guides the user to safety, while avoiding any dangerous zones.

These functions may change based on the active contexts and features. For instance, the application alerts the user that he is in danger when his position is inside a dangerous zones, which requires information about the position of the user and the location of the dangers. The guidance also changes based on the contexts and features: without any information about the safe places, it simply guides the user out of danger. When the application detects some internet connectivity (e.g. with LTE) and the location of the user (e.g. in Belgium), it provides a map that displays the dangers, safe places, and guidance visually. When the battery of the device is low, it disables the map representation, keeping only the textual representation in order to preserve battery power.

4.3 Modeling

There are many ways in which to design an ERS application with the functionality described in the previous section. In this section, we provide a model of such an ERS application.

The model is the one that we briefly introduced in the previous chapter (see Section 3.3.3): contexts and features are separated into two distinct models, namely the *Context Model* and the *Feature Model*. Each of these models are again split into two sub-models, a *configuration model* and a *dependency model*. The *Context Model* and the *Feature Model* are linked together by means of *Mapping* from particular contexts to particular features. We therefore split the whole model of the ERS application as follow:

- **Context Model:** Configuration & Dependency Sub-Models
- **Feature Model:** Configuration & Dependency Sub-Models
- **Mapping**

4.3.1 Context Model

Contexts are modeled in a dedicated ***Context Model***. This model is split into two sub-models: a *Context Configuration Model* (Figure 4.1) and a *Context Dependency Model* (Figure 4.2).

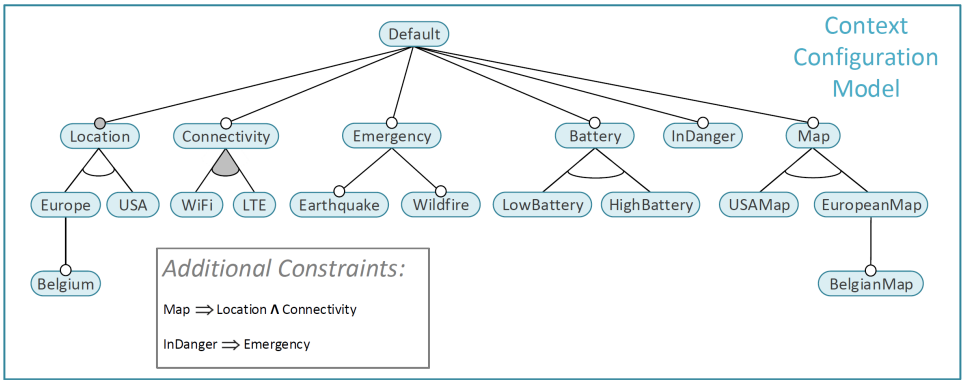


Figure 4.1 – Context Configuration Model of the ERS.

We briefly describe the most important contexts in the models:

- `Default` represents the default context, which is always active. It is used to associate contexts that are active when we initially run the application, such as the `Location` that is set to `USA` by default.
- `Location` is active when the location of the user is known[1].
- `Connectivity` is active when there is an internet connection, either through `WiFi` or `LTE` (non-exclusive).
- `Emergency` is active when an emergency is detected, such as an `Earthquake`.
- `LowBattery` is active when the application detects that the battery of the device is low. Although we will manually activate or deactivate this context in our simulation, we can assume that it is detected automatically by the Operating System of the device.
- `InDanger` states whether the user is in danger or not. This is a bit of a special context, as we will see that it is the application that will decide whether this context is active or not. The user cannot be "in-danger" if there is no emergency detected.
- `Map` states whether the application should use a map or not. It does so when it has `Connectivity` and `Location` that are both active. The map is centered on the location of the user, which is either on USA or on Europe.

[1]We consider a tiny world with just Europe and USA, which are of course disjoint.

The *Context Configuration Model* in Figure 4.1 states which sets of active contexts are valid. In particular, any ERS application must be aware of the `Location` of the user, and the location must either be `Europe` or `USA` (and in the case of europe, it can be refined to `Belgium`).

The *Context Dependency Model* in Figure 4.2 dictates the dynamic cascading and constraining relations between contexts. In particular, it says that when `Default` becomes active (which is the case when we initially launch the application), the contexts `USA` and `LTE` become active. As a result, `USAMap` becomes active by successive conjunctive, disjunctive and causal dependency relations.

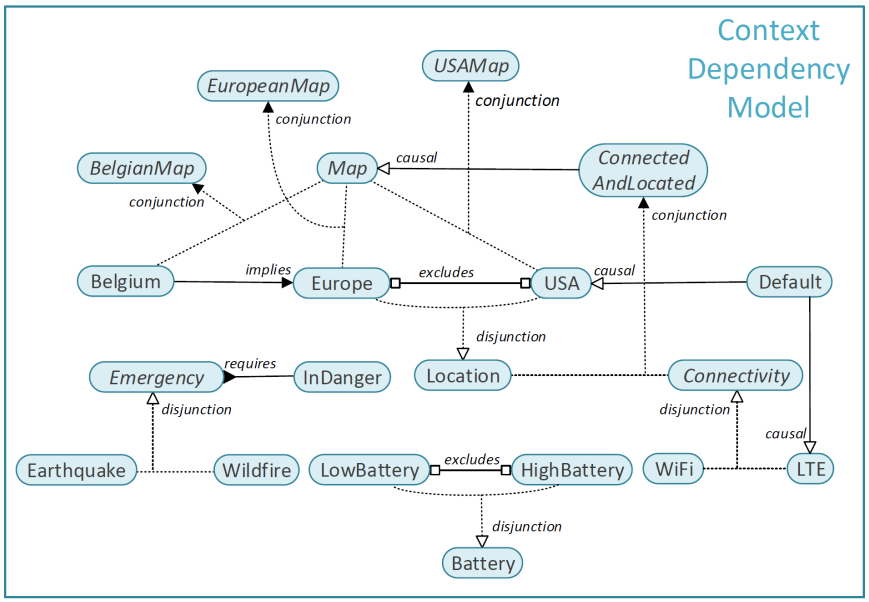


Figure 4.2 – Context Dependency Model of the ERS.

Some relations have a special handling, such as the ones that link `Battery`, `HighBattery` and `LowBattery` together: When the context `HighBattery` is active, and the devices detects that the battery becomes low, it will activate the context `LowBattery`. We can see that there will be a conflict due to the relation *excludes* between these two contexts: `HighBattery` is active, while `LowBattery` has a pending activation.

There are two ways to completely activate `LowBattery`: either we activate `LowBattery` and deactivate `HighBattery` simultaneously, or we define a conflict resolution strategy that will automatically deactivate `HighBattery` whenever we activate `LowBattery` (see *Attempt* resolution strategy in Section 3.3.3). We chose the second approach, which is explained in more details in the chapter *Implementation* (see Section 6.2.1)

4.3.2 Feature Model

Features are modeled in a dedicated ***Feature Model***. Same as for contexts, this model is split into two sub-models: a *Feature Configuration Model* (Figure 4.3) and a *Feature Dependency Model* (Figure 4.4).

We briefly describe the most important features in the models:

- `FDefault` represents the default feature, which is always active. It is used to associate features that are active when we initially run the application, such as `informGeneralInstructions` that is always active.
- `informUser` groups many features that are related to the function ***Inform*** of the application: it informs general instructions, detected emergencies, surrounding safe places, and guidance (textual representation of the feature `guideUser`, which is explained below). Also, it informs the emergency call, which depends on the location (911 in USA, 112 in Europe).
- `alertUser` groups features that are related to the function ***Alert*** of the application: it alerts when an emergency has been detected, or when the user is in danger (i.e. he is within a dangerous zone).
- `showMap` groups features that are related to the hidden function of displaying a map, which is the case when the context `Map` is active: it shows the location of the user (USA, Europe, Belgium) and displays some elements on the map (user's position, dangerous zones, safe places, guidance route).
- `guideUser` groups features that are related to the function ***Guide*** of the application. There are 3 types of algorithms to guide the user, based on the active feature: (1) it guides the user out of danger if he is inside a dangerous zone; (2) it guides the user to the closest safe place; (3) it guides the user to the closest safe place, while avoiding danger as much as possible. The chosen algorithm is based on the information that is provided by the application (only on emergencies, only on safe places, or both at the same time).

The *Feature Configuration Model* in Figure 4.3 states which sets of active features are valid. In particular, any ERS application must inform the general instructions "in case of emergency", such as providing the emergency call 112 if the user is in Europe (or 911 if he is in the USA). It contains many additional constraints to link same concerns that are scattered in multiple functions: in particular, it is inconsistent to have the feature `informGuidance` without `guideUser`. The other way around is valid, despite not showing any guidance (neither textually, nor visually).

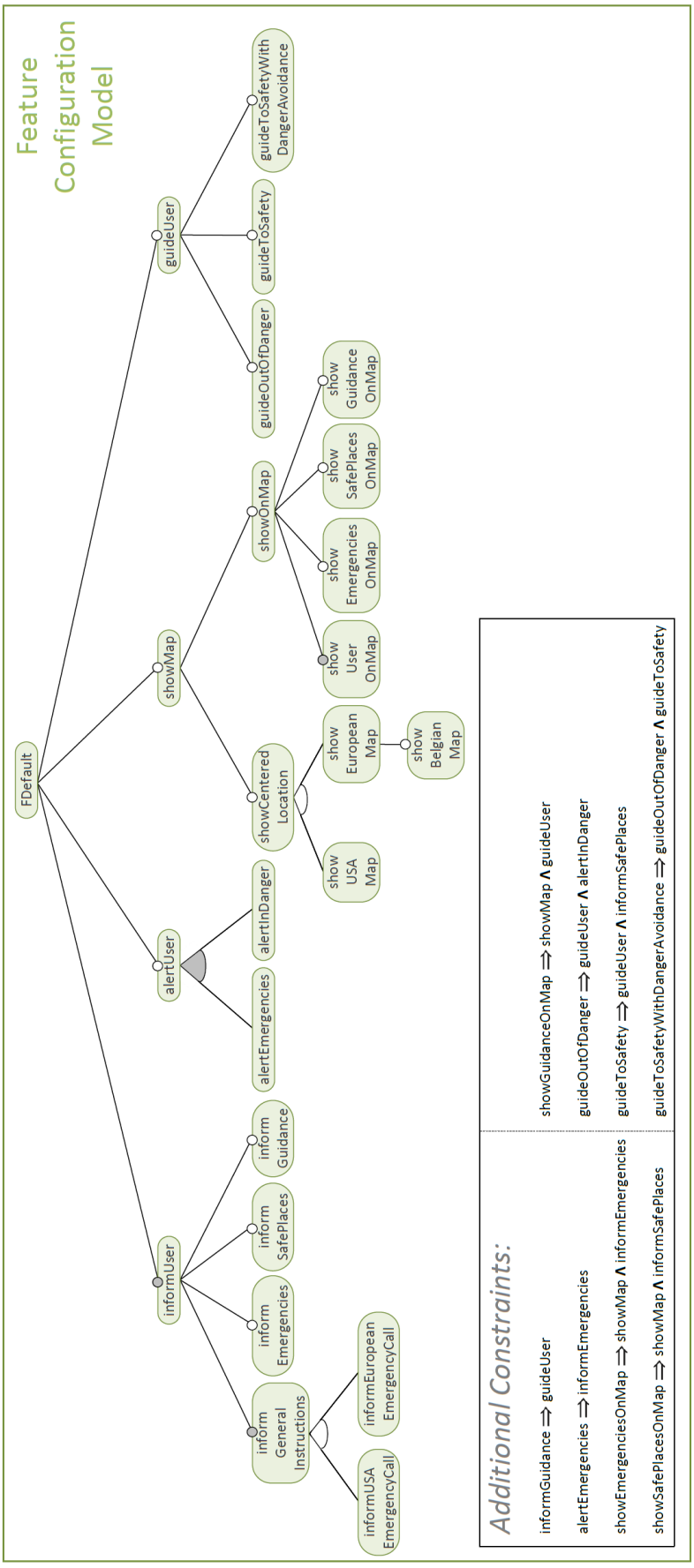


Figure 4.3 – Feature Configuration Model of the ERS.

The *Feature Dependency Model* in Figure 4.4 dictates the dynamic cascading and constraining relations between features. In particular, it says that when `FDefault` becomes active (which is the case when we initially launch the application), the feature `informGeneralInstructions` becomes active. Also, when the features `showMap` and `guideUser` are both active, it displays a visual map containing showing the guidance route (`showGuidanceOnMap`) on the map. We use a lot of conjunctive relations in order to have smart guidance and smart use of the map.

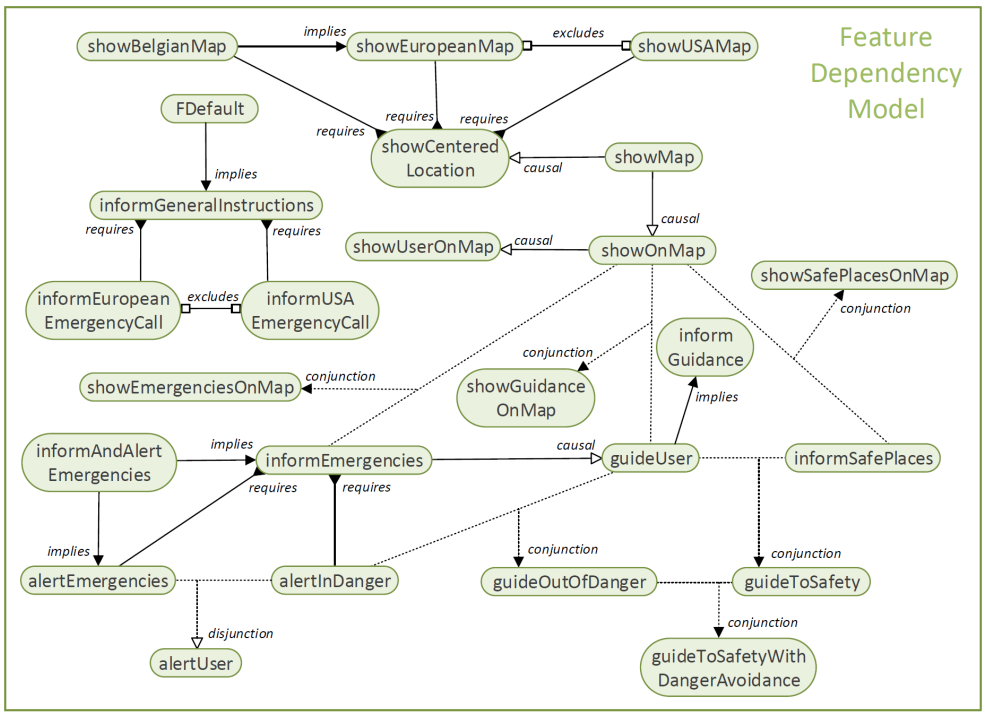


Figure 4.4 – Feature Dependency Model of the ERS.

4.3.3 Context-Feature Mapping

Figure 4.5 shows the mapping between contexts and features. Mapping is like a toggle relation. For instance, if the context `Emergency` is active and the feature `informAndAlertEmergencies` is inactive, then this feature becomes active: it alters the application so that it behaves by informing and alerting the user from surrounding emergencies and dangerous zones.

It is also the case with the context `Map` and the feature `showMap`, except when the context `LowBattery` is active, as disablers have priority over enablers: when the context `LowBattery` is active and the feature `showMap` is active, this feature becomes inactive in order to save battery power.

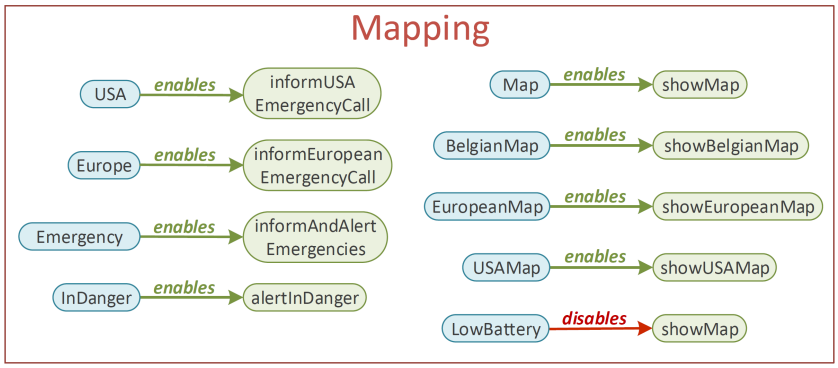


Figure 4.5 – Context-Feature Mapping of the ERS.

4.4 Scenario

We have concocted a scenario to test our ERS application with the previously defined model. We will conduct the scenario by following these steps:

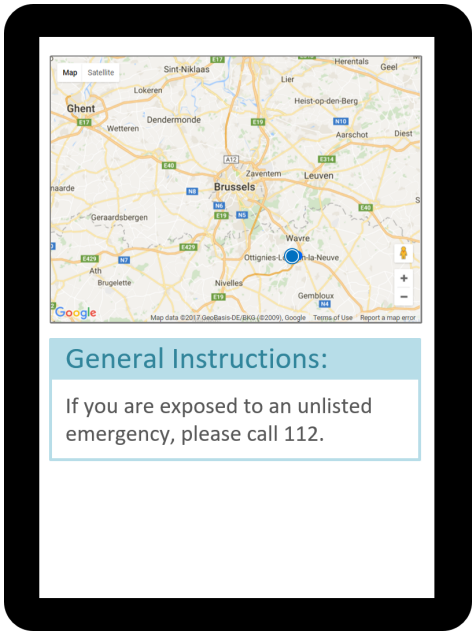
1. Initially, the ERS Application informs the user with general instructions. Our device detects that we are in Belgium so that the application displays the map of Belgium with the position of the user in blue, since we also have connectivity by LTE (Figure 4.6a).

2. After a while, we moved to Brussels, and unfortunately an earthquake is detected with the center of Brussels being its epicenter. The application alerts us of the earthquake and that we are in danger, in addition to guiding us out of the dangerous zone, visually on the map and textually with guidance instructions (Figure 4.6b).

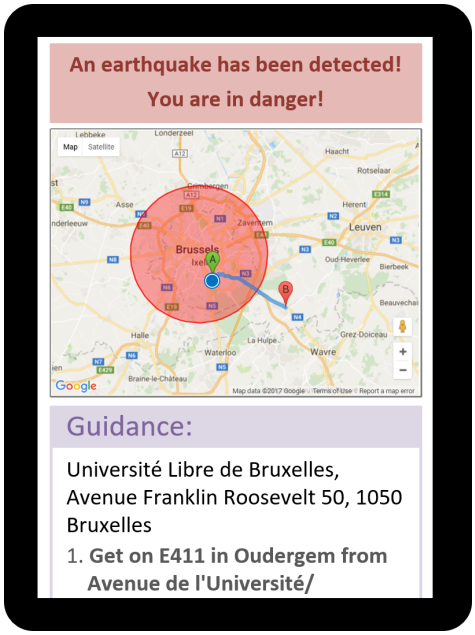
3. We activate the feature to be informed of surrounding safe places, so that we are guided to the nearest ones. Ottignies-Louvain-la-Neuve is the nearest safe place, but the application detected a wildfire on our path to Wavre. The closest safe place, while avoiding as much danger as possible, is an hospital at Leuven, so the application guides us to this safe place (Figure 4.6c).

4. The device detects that the battery capacity becomes low, so it disables the map representation. We disable the guidance manually to reduce battery power even further (Figure 4.6d).

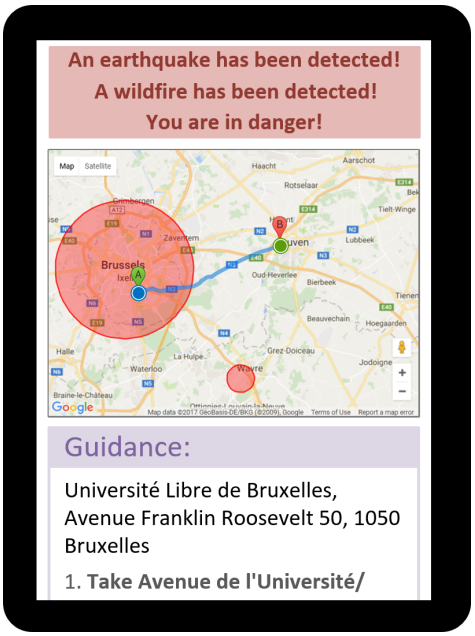
We have implemented this case study in our framework. A glimpse of how we have accomplished this task is explained in Section 6.3.



(a) From the default behavior of the application, the context `Belgium` is activated.



(b) Contexts `Earthquake` and `InDanger` are activated.



(c) Context `Wildfire` and feature `informSafePlaces` are activated.

(d) Context `LowBattery` is activated (completely), feature `guideUser` is deactivated.

Figure 4.6 – Screen-shots of the ERS.

Chapter 5

Architecture

In this chapter, we describe the architecture of our framework.

5.1 Global View

The global architecture of our framework is illustrated in Figure 5.1. It shows its work-flow when a context-aware application wishes to activate or deactivate a context or a feature.

Each rectangle represents a step in the work-flow, and each arrow represents information that is transmitted from one step to another one. Red rectangles are ***user-startable*** steps, meaning that the flow can start from any of these steps by the context-aware application itself (here, user refers to the developer that uses our framework, not the user of the application!) In particular, when an application desires to activate a context, it provides information to the step `Context Activation`.

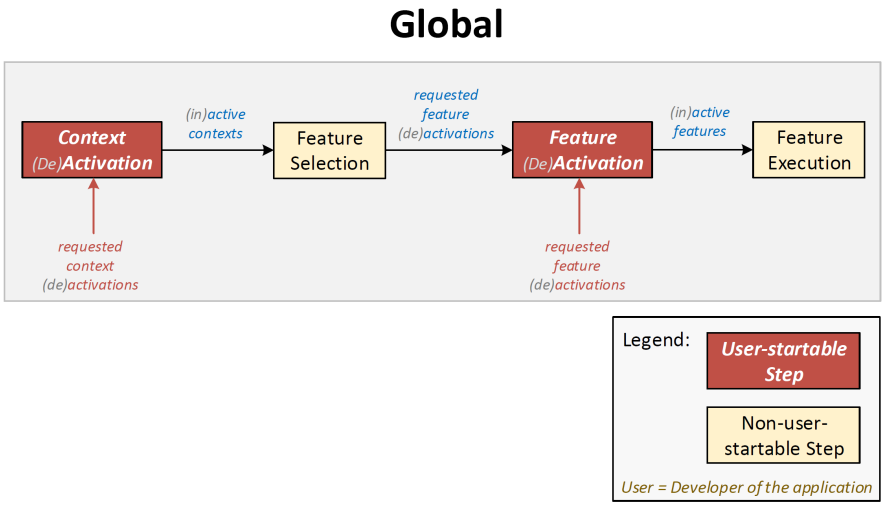


Figure 5.1 – Global Architecture of the Framework.

The activation or deactivation of contexts may lead to the activation or deactivation of several features. This process is split into 3 steps: (1) it

selects the features to activate or deactivate based on the active contexts (`Feature Selection`); (2) it derives the features to activate or deactivate (`Feature Activation`); (3) it executes the active features so that they run within the context-aware application (`Feature Execution`).

This architecture is heavily inspired by the Handling layer of the layered COP architecture (see Section 2.2.2). We describe in more details how each of these steps work:

- When the context-aware application wishes to activate or deactivate some contexts, it provides the requests to activate or deactivate these contexts to the component `Context Activation`. From these requests, it activates or deactivates several contexts, so that it gets the list of all active or inactive contexts.
- From the list of all currently active or inactive contexts, the component `Feature Selection` deduces the features for which their activation or deactivation are directly influenced by the active or inactive contexts. It outputs the requests to activate or deactivate the corresponding features.
- Then, from the requests to activate or deactivate the features, the component `Feature Activation` handles the activation or deactivation of the features, in a similar fashion to the activation or deactivation of contexts from the component `Context Activation`, so that it gets the list of all active or inactive features.
- Finally, from the list of all active or inactive features, the component `Feature Execution` handles the execution of features, i.e. it applies changes to the application so that it possesses the active features.

As we stated before, red-colored rectangles are ***user-startable*** steps of the work-flow: an application using this framework can request the activation or deactivation of multiple contexts, which will provide this information to the component `Context Activation`. Similarly, the same application can request the activation or deactivation of multiple features, which will provide this information to the component `Feature Activation`.

5.2 Activation

The components `Context Activation` and `Feature Activation` are almost similar, the sole difference being that one deals with contexts and the other deals with features. As a result, we explain only the reasoning of the component `Context Activation`. Figure 5.2 illustrates the reasoning for both of these components.

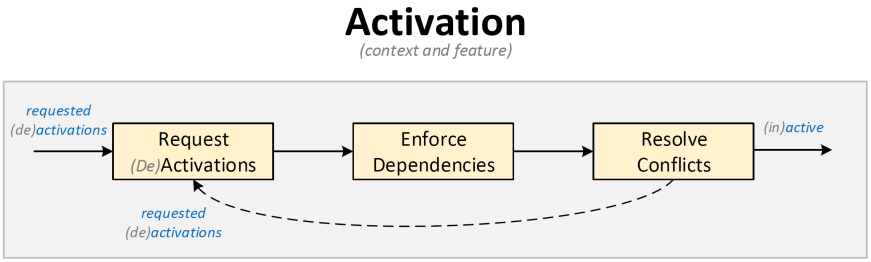


Figure 5.2 – Tasks of the Components `Activation` (Context and Feature).

Request (De)Activations When the component receives the requests to activate or deactivate some contexts, it registers these requests on the *Context Model*. The model is changed so that it tracks these requests, but it doesn't change the state of the corresponding contexts.

Enforce Dependencies When the model has registered the requests, it must apply them by enforcing the dependency relations between contexts, so that the requested activation or deactivation of the contexts results in a situation in which some contexts become active or inactive.

Resolve Conflicts Usually, the *Context Model* is stable after enforcing the dependencies. A model is said to be stable when it has no conflict. A conflict occurs when a context couldn't become active or inactive, although it should have been. For instance, a context couldn't become active because another context is active and there is an exclusion relation between both contexts. To ensure that the model is stable, it must not have any conflict. When there is such a conflict, it tries to resolve it, which might request for more activation of deactivation of contexts. When this module resolved all conflicts (or choose to do nothing with the conflicts), it outputs the list of all currently active or inactive contexts.

For our case study, we have implemented a conflict resolution strategy that *attempts* to resolve the conflicts automatically, otherwise it *rolls back* the active state of the activatable entities, i.e. it reverts to the most-recent situation without any conflict. We show it in details in Section 6.2.1.

Chapter 6

Implementation

We describe in this chapter how we have implemented the framework. First, we give the structure of the code and describe the main components. Second, we show the workflow of the code when we activate a context, and give details on each step. Third, we show how to use the framework in order to design our ERS application.

6.1 Structure of the Code

The code contains many classes and objects, therefore we split its structure in 3 parts: the core components of the architecture, the components related to the dependency model, and the components related to the configuration model.

6.1.1 Core Components

Figure 6.1 shows the structure of the code with the core components of the framework. We briefly describe those components:

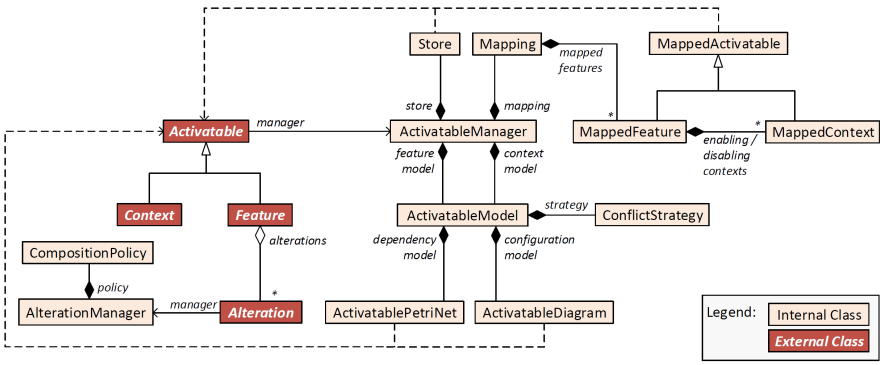


Figure 6.1 – Structure of the Code (Core Components).

Activatable Entities that can be activated are instances of this class. Contexts and Features are specialized instances of the class Activatable. An activatable entity has a `name`, can be activated or

deactivated, and it has a reference to a system-wide `manager` (see description of ***ActivatableManager*** below).

Context This is a specialized instance of Activatable. Contexts are reified as instances of this class.

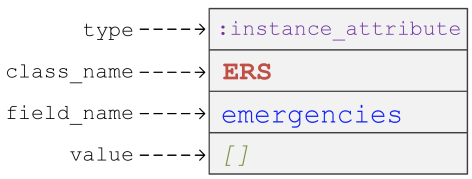
Feature This is a specialized instance of Activatable. Features are reified as instances of this class. It enhances the behavior of a basic activatable entity with `alterations`, which let this feature alter the code of the application when it becomes active.

ActivatableManager This is a singleton class that manages anything that is related to activatable entities. It holds a `store` containing all registered entities in the system, the `context_model` and `feature_model` (see description of ***ActivatableModel*** below), in addition to the `mapping` between particular contexts and features (see description of ***Mapping*** below). Most tasks of the architectural step *Feature Execution* are processed by this component.

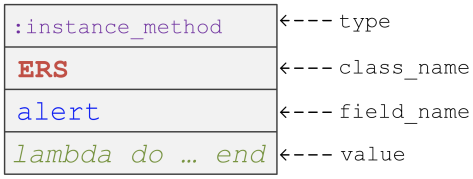
ActivatableModel The *Context Model* and *Feature Model* are implemented as instances of this class. Each of these models holds two sub-models: a `configuration_model` (see Section 6.1.2) and a `dependency_model` (see Section 6.1.3). The model also holds the `strategy` that is used in order to resolve any detected conflicts.

Mapping Mapping between contexts and features are handled by this class. It stores the list of all `mapped_features`, each of them holding the `enabling_contexts` and `disabling_contexts`.

Alteration Alterations are instances of this class. An instance of Alteration has a `type`, refers to a `class_name` and a `field_name` and a `value`. Figure 6.2 illustrate two examples of alterations from our ERS application: (a) on the left, it shows an alteration on the class ERS which sets an attribute `emergencies` that is an empty array; (b) on the right, it shows an alteration on the class ERS which alters the method alert, such as to warn the user of nearby emergencies (we show the code of both alterations later, see Listing 6.15). It has also a reference to its system-wide `manager` (see description of ***AlterationManager*** below).



(a) Alteration of type `:instance_attribute` on class `MyClass` and attribute `i`.



(b) Alteration of type `:instance_method` on class `MyClass` and method `to_s`.

Figure 6.2 – Visual Examples of Alterations.

AlterationManager This is a singleton class that manages the insertion of alterations into our context-aware application. It holds the `applied_alterations`, a `policy` for the composition of alterations with the same `class_name` and `field_name`, and a special stack called `proceeds` to handle the *proceed* mechanism in COP (see Section 2.2.4 on Method Pre-Dispatch and Proceed).

The classes **Activatable**, **Context**, **Feature**, and **Alteration** are external, meaning that our ERS application must explicitly and solely make use of these classes. Although Mapping is an internal class, the application can make changes on it by means of the class Activatable, as illustrated in Figure 6.3, showing the sequence diagram to add a mapping between a context c and a feature f of type `:enabler`.

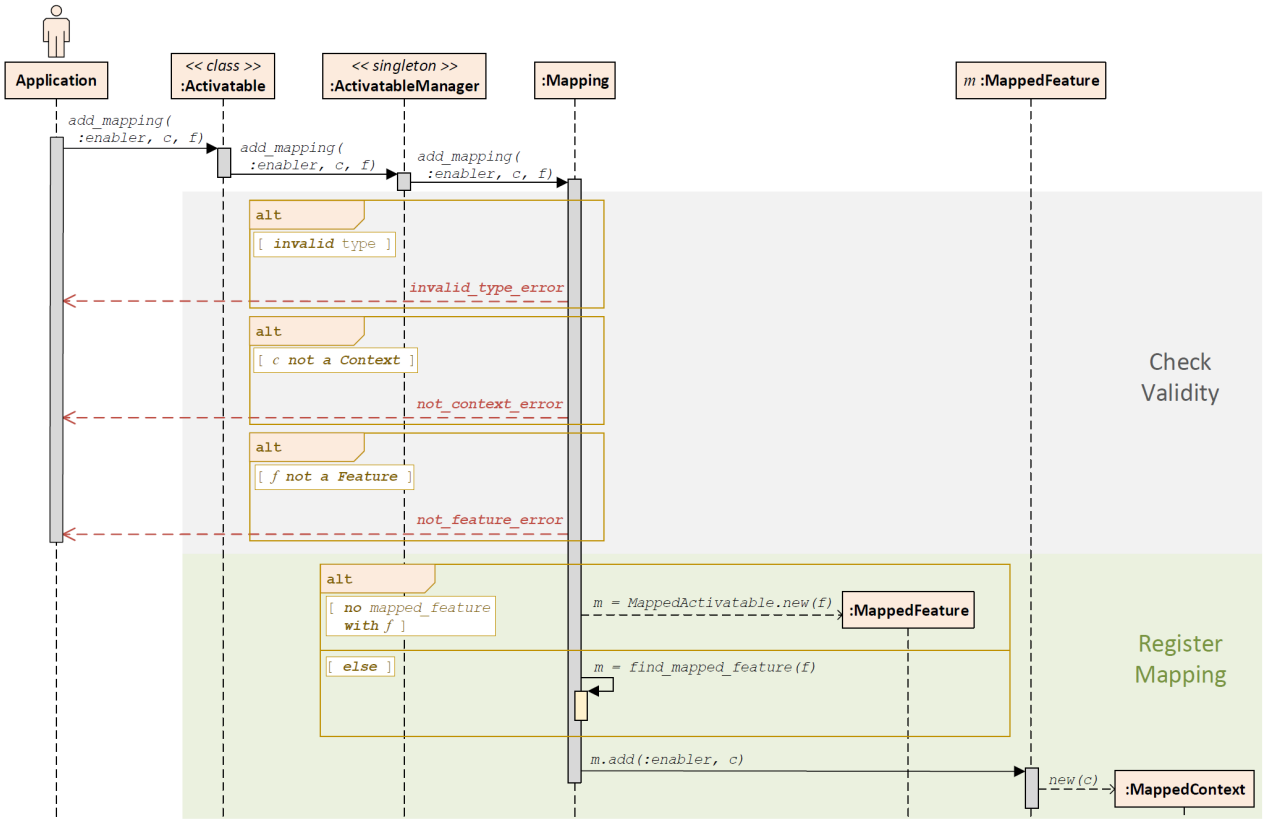


Figure 6.3 – Sequence Diagram to Add Mapping (using the class `Activatable`).

6.1.2 Configuration Model

Figure 6.4 shows the structure of the code with the components related to the configuration sub-model of the *Context Model* and the *Feature Model*.

The main class is `ActivatableDiagram`. It holds the `relations` between activatable entities (e.g. there is a `:mandatory` relation between contexts `Default` and `Location`), the actual representation of these relations as `expressions` (e.g. the previous mandatory relation is translated

as `Default` $\Longrightarrow$ `Location`), and a set of `algorithms` to translate these relations in terms of expressions.

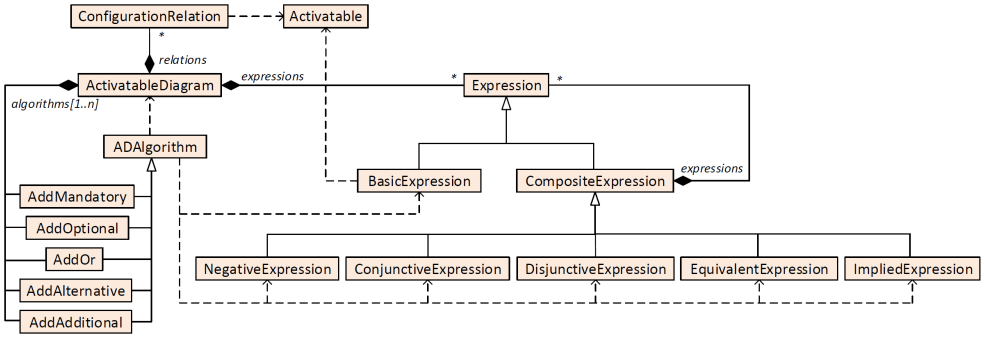


Figure 6.4 – Structure of the Code (Configuration-Related Components).

Figure 6.5 illustrates how we have implemented the visual representation of the configuration model in the framework: a relation is an inline representation of a hierarchical link (e.g. `Location` is a *mandatory* context from `Default`) or an additional constraint (e.g. `InDanger` $\Longrightarrow$ `Emergencies`).

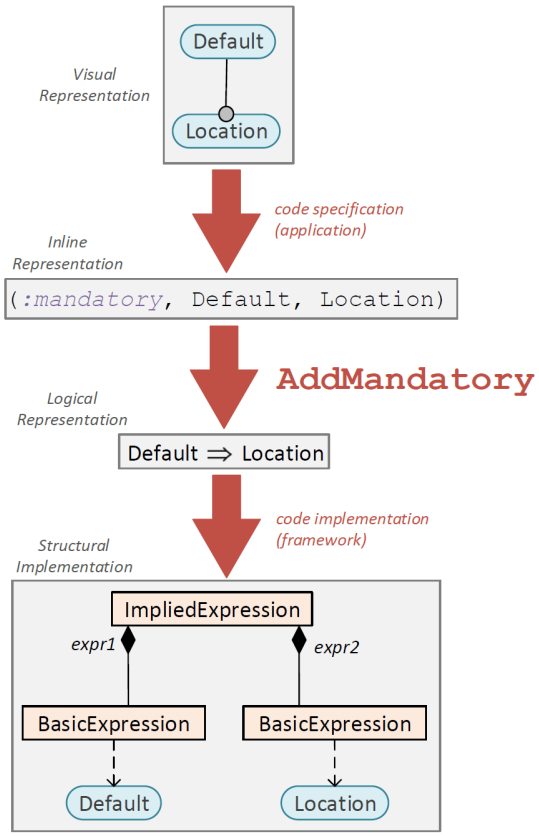


Figure 6.5 – Implementation of a Configuration Relation (*mandatory* relation).

In the case of a hierarchical link such as the mandatory relation illustrated in this figure, it can be translated into a logical representation, which is equivalent to how we define additional constraints (`Default` $\Longrightarrow$ `Location`). The logical representation is turned into code using the *Interpreter Design Pattern*, in order to have the resulting structural implementation. To do so, it uses the algorithm `AddMandatory`, which will produce an `ImpliedExpression` corresponding to the translation of the *mandatory* relation into expressions.

6.1.3 Dependency Model

Figure 6.6 shows the structure of the code with the components related to the dependency sub-model of the *Context Model* and the *Feature Model*.

The main class is `ActivatablePetriNet`. It holds the `relations` between activatable entities (e.g. (`:exclusion` between `LowBattery` and `HighBattery`), the actual representation of these relations as Petri Net's `elements` (similarly to CoPN in Section 2.3.4), and a set of `algorithms` to translate these *relations* into *elements*.

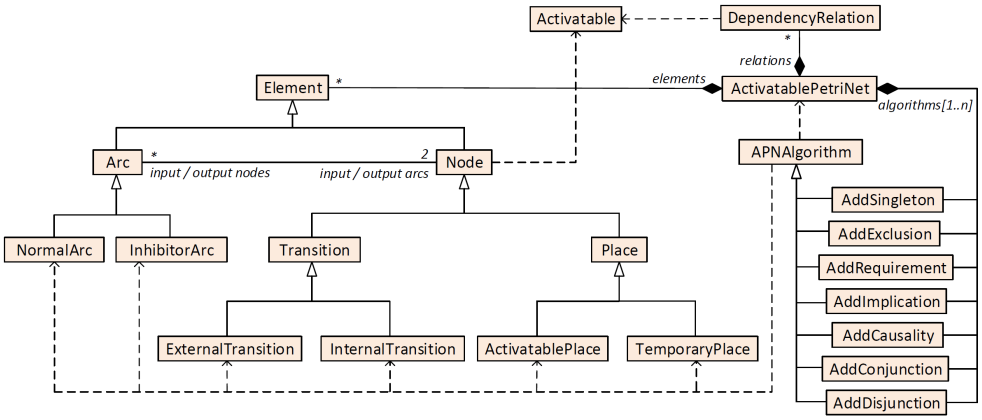


Figure 6.6 – Structure of the Code (Dependency-Related Components).

Figure 6.7 illustrates how we have implemented the visual representation of the dependency model in the framework: a relation is an inline representation of a dependency relation. It can be translated into a logical representation, here with CoPN. This representation is implemented with elements turned into objects, here the two added inhibitor arcs in the case of an exclusion relation, using the algorithm `AddExclusion`. The elements from the singletons have already been built and stored when we have created both contexts, using the algorithm `AddSingleton`.

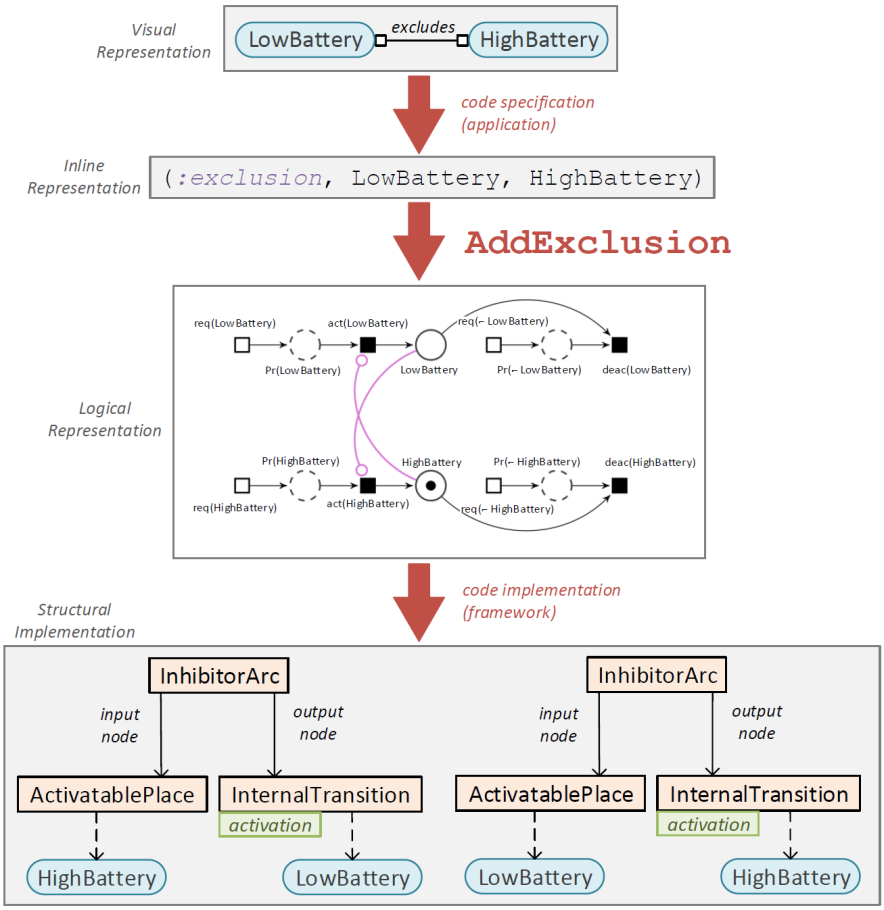


Figure 6.7 – Implementation of a Dependency Relation (*exclusion* relation).

6.2 Workflow of the Code

Figure 6.8 shows the sequence diagram when we activate a context c:

1. To activate the context c, the application sends the message `activate` to c.
2. The context c submits the request to activate itself to its manager (`ActivatableManager`).
3. The manager fowards the message to the *Context Model*, which will activate or deactivate the contexts based on the requests, the dependencies, and the strategy to handle conflicts. As a result, it returns the list of all active and inactive contexts.
4. From the list of active and inactive contexts, the manager asks the mapping to select the features. The mapping returns the features to activate and the features to deactivate.

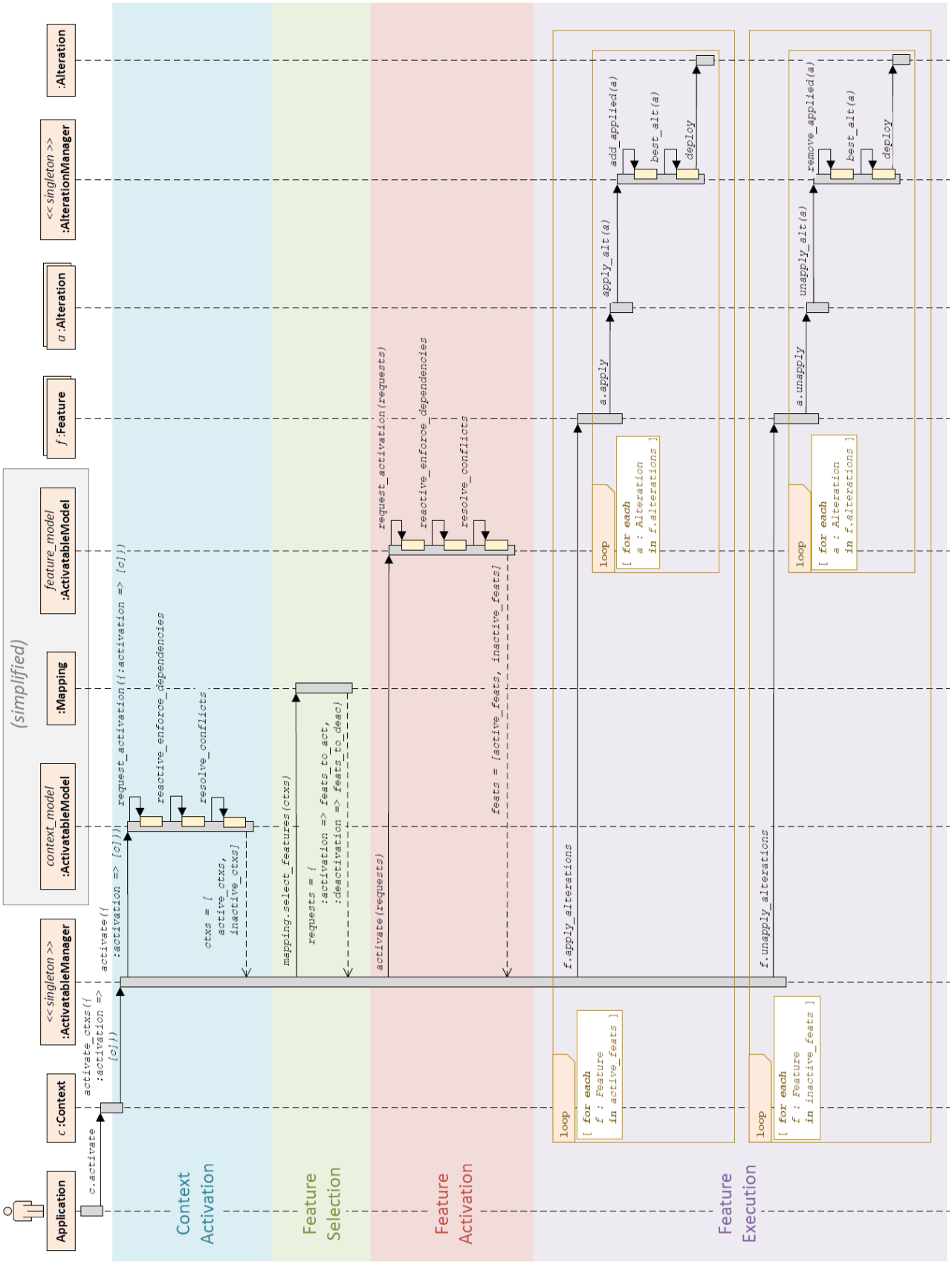


Figure 6.8 – Sequence Diagram for the Activation of a Context c.

5. The manager forwards this list of features to (de)activate to the *Feature Model*, which will perform similar tasks to the Context Model but on features. Similarly to the Context Model, it returns the list of all active and inactive features.

6. Then, each active feature applies their alterations, which consists of sending the message `apply` to all of their alterations.

7. Alterations send the requests to apply themselves to their manager (`AlterationManager`), so that it can track applied alterations and deploy the best one, according to the composition policy that has been chosen by the application (see Section 6.2.3 for a detailed explanation of the composition policy).

8. An analogous process is performed by inactive features, except that they send to all of their alterations the message `unapply`.

This sequence diagram has been simplified for *Context Model*, *Feature Model*, and *Mapping*: it does not show how any of the models interacts with their *configuration* and *dependency* sub-models. Furthermore, it does not show how the mapping interacts with its mapped features in order to select the appropriate features to activate or to deactivate.

We explain these mechanisms separately in the following subsections (from 6.2.1 to 6.2.3).

6.2.1 Context and Feature Activation

As we showed in the Section 5.2, when the Context or Feature Model receives the message to activate or deactivate some activatable entities, it performs 3 tasks: (1) it applies the requested activation or deactivation on the model; (2) it enforces the dependencies between the entities; (3) it resolves any detected conflicts. We explain these tasks separately below.

Request (De)Activations

Listing 6.1 shows how it applies the requested activation or deactivation of the entities into the dependency model: for each type of request (`:activation` or `:deactivation`), it fires the external transitions of all activatable entities that are in the request. For instance, if the Context Model receives the request `{:activation => [lte]}`, it fires the external transition of sub-type `:activation` of the context `lte`.

```
class ActivatablePetriNet
    def request_activations(requests)
        requests.each do |type, acts|
            acts.each do |act|
                find_external_transition(type,
                    act).fire
            end
        end
    end
end
```

Listing 6.1 – Request (De)Activations (Context and Feature Model).

Reactive Enforce Dependencies

Listing 6.2 shows how it enforces the dependencies after applying the requested activation or deactivation of the entities: if there are internal transitions that are enabled in the dependency model, it selects one at random and fires it. It continues until there is no more internal transition that is enabled. The reason why we select at random is to avoid *infinite loops* [4], which is a limitation that we discuss in the next chapter (see Section 7.1).

```
class ActivatablePetriNet
    def reactive_enforce_dependencies
        its = find_enabled_int_transitions
        while not its.empty? do
            its.sample.fire
            its = find_enabled_int_transitions
        end
    end
end
```

Listing 6.2 – Reactive Enforce Dependencies (Context and Feature Model).

Resolve Conflicts

Listing 6.3 shows how we resolve the conflicts in our case study. We use a concrete example to understand the purpose of this code: let's assume that the context `HighBattery` is active. When the device detects that the battery power is low, it activates the context `LowBattery`. Due to the exclusion dependency relation between these two contexts, the context `LowBattery` could not become active, which results to a conflict.

This conflict is resolved as followed: the function `can_attempt_alternative` looks for conflicts having a particular pattern, which can possibly be re-

solved by `attempt_alternative`. The conflict that results from the activation of context `LowBattery` can be resolved with `attempt_alternative`: it detects that this context couldn't become active, because `HighBattery` is active. However, there is an *alternative* configuration relation between `LowBattery` and `HighBattery`, so the function `attempt_alternative` will take measures to resolve the conflict. More precisely, it will request the deactivation of `HighBattery`, so that `LowBattery` can completely become active. If there are conflicts that cannot be resolved by `attempt_alternative`, the framework forces the model to roll back.

```
class ConflictStrategy
    def resolve_conflicts
        cs = @model.get_conflicts
        if not cs.empty?
            if can_attempt_alternative?(cs)
                attempt_alternative(cs)
            else
                rollback
            end
        end
    end
end
```

Listing 6.3 – Resolve Conflicts (Context and Feature Model).

When the method `attempt_alternative` resolve the conflicts by requesting the deactivation of some contexts, it corresponds to the retroactive dashed arrow of Figure 5.2 from the component *Resolve Conflicts* and *Request (De)Activations*. There are some flaws to this design approach: in particular, it makes the occurrence of some conflicts intentional. We discuss this issue in the next chapter (see Section 7.1).

6.2.2 Feature Selection

When the Context Model has changed the active state of the contexts, there is a selection of the features to activate or to deactivate. This selection depends on the currently active contexts, and it is handled by the component `Mapping`.

Listings 6.4 and 6.5 show how the selection of the features is handled by `Mapping`: for all mapped features, it builds the requests for the activation or deactivation of the features. Each mapped feature tells whether it should request its activation or deactivation (or no change at all), based on the active contexts. The code of `MappedFeature` follows the definition of enablers and disablers from Section 3.3.3.

```
class Mapping
   def select_features(ctxs)
      requests = {
         :activation   => [],
         :deactivation => []}
      @mapped_features.each do |m|
         req = m.select_feature(ctxs)
         if not req.nil?
            append_request(requests, req)
         end
      end
      requests
   end
end
```

Listing 6.4 – Select Features From Active Contexts (`Mapping`).

```
class MappedFeature
   def select_feature(ctxs)
      e = any_enabling_active?(ctxs)
      d = any_disabling_active?(ctxs)
      if e and not d
            and @activatable.inactive?
         {:activation => [@activatable]}
      elsif d and @activatable.active?
         {:deactivation => [@activatable]}
      elsif not e and not d
            and @activatable.active?
         {:deactivation => [@activatable]}
      else
         nil
      end
   end
end
```

Listing 6.5 – Select Features From Active Contexts (`MappedFeature`).

6.2.3 Feature Execution

We show here how features are executed when they become active or inactive. As shown in Figure 6.8, when we have the list of all active or inactive features from the Feature Model, we must *apply* the alterations of all *active* features (respectively, *unapply* the alterations of all *inactive* features). Each feature possesses the list of all its alterations, and each alteration can be applied by the features.

Composition Policy

Listing 6.6 shows how `AlterationManager` handles the composition of the alterations, that is how it chooses which alteration to deploy when there are multiple alterations on the same `class_name` and `field_name`. It operates similarly to *Method Pre-Dispatch* (see Section 2.2.4): whenever an alteration is (un)applied, `AlterationManager` builds an sorted chain of alterations with the same `class_name` and `field_name` of the recently (un)applied alteration. From this chain, it deploys the best alteration, i.e. the first one in the chain.

```
class AlterationManager
    def apply(alteration)
        add_applied(alteration)
        best_alt(alteration).deploy
    end

    def best_alt(alteration)
        a = alteration_chain(alteration).first
        if a.nil?
            clear_field(alteration)
        end
        a
    end

    def alteration_chain(alteration)
        class_name = alteration.class_name
        field_name = alteration.class_name
        alts =
          @applied_alterations.find_all do |a|
            a.alters?(class_name, field_name)
          end
        @policy.do_sort(alts)
    end
end
```

Listing 6.6 – Composition Policy of Alterations (`AlterationManager`).

The policy to compose the alterations is defined in `CompositionPolicy`, as illustrated in Listing 6.7. It sorts out alterations that have similar attributes (`type`, `class_name`, and `field_name`), according to its policy.

We choose to sort out alterations based on the activation age of their features, that is the time that has passed from receiving their last token in the Petri Net until now. This is how the method `global_sort` sorts out the alterations: younger features have priority over older ones.

```
class CompositionPolicy
   def do_sort(alterations)
      chain1 = global_sort(alterations)
      chain2 = custom_sort(chain1)
      chain2
   end

   def global_sort(alterations)
      # Sort 'alterations' by corresponding
      # features' activation age.
   end

   def custom_sort(alterations)
      # Refines 'alterations' with
      # a custom sort.
      # e.g. ':show_belgian_map' comes
      #      before ':show_european_map'
   end
end
```

Listing 6.7 – Composition Policy of Alterations (`CompositionPolicy`).

The sorting by activation age is enough to have a total order among alterations, but we would like to have some older features that are more specific than newer ones. In particular, the feature `showBelgianMap` is always older than the feature `showEuropeanMap` because of the implication dependency relation. This is not the desired order, because the feature `showBelgianMap` is finer than `showEuropeanMap`. To solve this problem, we specify a custom order between these two features, so that `showBelgianMap` has higher priority than `showEuropeanMap`. This is how the method `custom_sort` refines the array of alterations produced by `global_sort` (we show later in this chapter how to set a custom order between two features, in Listing 6.16).

Code Deployment

Listing 6.8 shows how the framework deploys alterations on the application. For the moment, we restrict our explanation with alterations of type `:instance_attribute`, as alterations of type `:instance_method` must handle the `Proceed` mechanism from COP, which is explained just after this part.

When the `AlterationManager` decides to deploy an alteration of type `:instance_attribute`, it adds the attribute `field_name` to the class `class_name`. Existing objects from this class will have the newly created attribute assigned to `value`.

```
class Alteration
   def deploy
      if @type == :instance_attribute
         self.deploy_instance_attribute
      elsif @type == :instance_method
         self.deploy_instance_method
      end
   end

   def deploy_instance_attribute
      c = Object.const_get(@class_name)
      c.send(:attr_accessor, @field_name)
      ObjectSpace.each_object(c) do |obj|
         obj.send(:instance_variable_set,
            "@#{field_name}", @value)
      end
   end
end
```

Listing 6.8 – Deploy Alterations.

Newly created objects from this class will also have this attribute, because it is defined in the definition of the class. However, they will not have the value defined in the corresponding alteration, because the framework only assigns the value to existing objects at the moment it deploys the alteration. We can remedy this problem by altering the method `initialize` of the class, so that it correctly assigns the value to this attribute. We briefly discuss this limitation in the Section 7.1.

`Proceed` Mechanism

Alterations of type `:instance_method` support the `Proceed` mechanism of COP. Listing 6.9 shows how the framework adds the keyword `proceed`, by simply adding a method `proceed` to the the altered class. This method asks `AlterationManager` to handle the proceed mechanism.

```
class Alteration
   def initialize
      # initialize attributes
      Object.const_get(@class_name).
         class_eval("
            def proceed(*args)
               Alteration.manager.proceed(
                  self, *args)
            end")
   end
end
```

Listing 6.9 – `Proceed` Mechanism (`Alteration`).

`AlterationManager` handles the `Proceed` mechanism by following the technique Method Pre-Dispatch, and tracks called altered methods by means of an ADT named `proceeds`. Figure 6.9 illustrates how proceeds is used to track called altered methods: it is a stack in which entries are alterations.

Whenever an altered method is called, the corresponding alteration is pushed in the stack before executing the altered method. During the execution of the method, the size of the stack might fluctuate if the current altered method calls other altered methods.

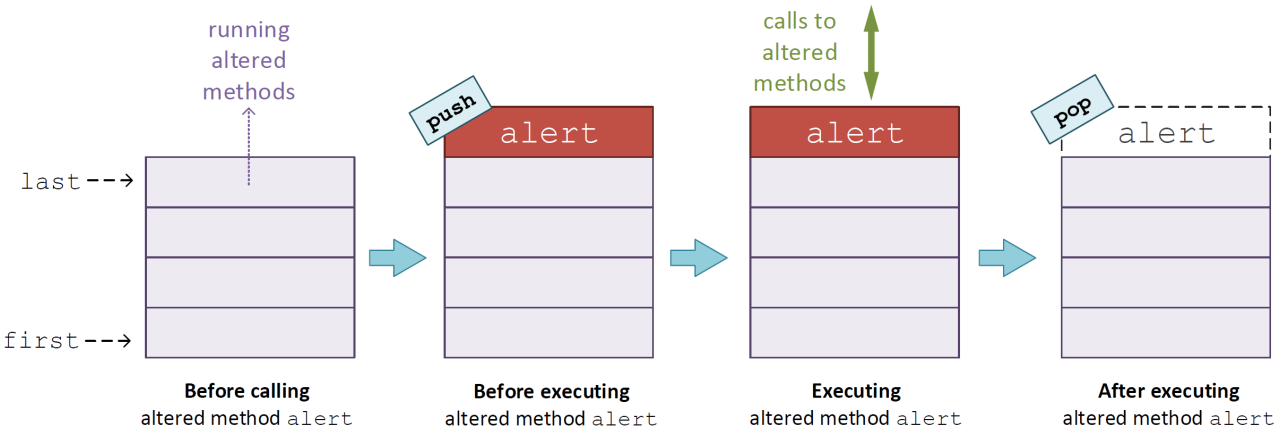


Figure 6.9 – Tracking of Called Altered Methods (`Proceeds`).

Listing 6.10 shows how AlterationManager handles the call `proceed` from any altered methods: it looks at the last called altered method and selects the next alteration in the chain for deployment. The deployment is just temporary: after the message `proceed`, we expect to have the same deployed alteration as before the call (`curr`), so we must deploy it back before forwarding the result of the call `proceed`.

```
class AlterationManager
   def proceed(obj, *args)
      curr = @proceeds.last
      self.alteration_after(curr).deploy
      res = obj.send(curr.field_name, *args)
      curr.deploy
      res
   end
end
```

Listing 6.10 – `Proceed` Mechanism (`AlterationManager`).

6.3 Our Context-Aware Application

In this section, we show some samples of the code of our ERS application.

Listing 6.11 creates the context `Emergency` and the feature `showMap`.

```
emergency = Context.new(:emergency)
show_map  = Feature.new(:show_map)
```

Listing 6.11 – ERS Application: Create Contexts and Features.

Listing 6.12 defines some relations between contexts:

- It adds an *optional* parent-child relation between the context `Default` and the context `Emergency`, in the Context Configuration Model.
- It adds an *alternative* relation between `LowBattery` and `HighBattery` in the Context Configuration Model, where `Battery` is the parent of the former contexts.
- It adds the following *additional* constraint in the Context Configuration Model: `Map` $\implies$ (`Location` $\wedge$ `Connectivity`).
- It adds a *causality* relation between the context `Default` and the context `USA`, from `Default` to `USA`.

```
Context.add_configuration_relation(
   :optional,
   Context.default,
   emergency )
Context.add_configuration_relation(
   :alternative,
   battery,
   low_battery,
   high_battery )
Context.add_configuration_relation(
   :additional,
   [  :implication,
      map,
      [  :conjunction,
         location,
         connectivity ] ] )
Context.add_dependency_relation(
   :causality,
   Context.default,
   usa )
```

Listing 6.12 – ERS Application: Define Relations Between Entities.

Listing 6.13 activate or deactivate some contexts: (1) it activates the context `WiFi`; (2) it simultaneously activates the context `Europe` and deactivates the context `USA`.

```
wifi.activate
Context.activate({
    :activation => [europe],
    :deactivation => [usa] })
```

Listing 6.13 – ERS Application: Activate Contexts.

Listing 6.14 defines some mapping between particular contexts and features: (1) it maps the context `Map` and the feature `showMap` together, so that `Map` is an *enabling* context of `showMap`; (2) it maps the context `LowBattery` and the feature `showMap` together, so that `LowBattery` is a *disabling* context of `showMap`.

```
Activatable.add_mapping(
    :enabler,
    map,
    show_map )
Activatable.add_mapping(
    :disabler,
    low_battery,
    show_map )
```

Listing 6.14 – ERS Application: Define Context-Feature Mapping.

Listing 6.15 adds some alterations to existing features:

- When the feature `informEmergencies` becomes active, it alters the class `ERS` of the application, so that instances of this class possess an attribute `emergencies`. When the alteration is deployed, the initial value of this attribute is an array containing a single emergency, an earthquake.
- When the feature `alertEmergencies` becomes active, it alters the class `ERS` of the application, so that instances of this class possess a method `alert` that prints a message to warn users of nearby emergencies.

```
inform_emergencies.add_alteration(
    :instance_attribute,
    :ERS,
    :emergencies,
    [ Earthquake.new ] )
alert_emergencies.add_alteration(
    :instance_method,
    :ERS,
    :alert,
    lambda do
        s = ""
        @emergencies.each do |e|
            s += "#{e.type} detected nearby\n"
        end
        s
    end )
```

Listing 6.15 – ERS Application: Define Alterations to Existing Features.

Listing 6.16 defines a custom order between two features for the composition of alterations: it states that the feature `showBelgianMap` is more specific than `showEuropeanMap`, so that it shows the map of Belgium even though `showEuropeanMap` has been activated more recently than `showBelgianMap` (because of the *implication* relation between those features).

```
Alteration.set_custom_order(
    :show_belgian_map,
    :show_european_map )
```

Listing 6.16 – ERS Application: Customize Alteration Composition.

The listings show that the interactions with the framework are quite basic. We discuss how we could improve these interactions in the next chapter (see Section 7.2).

Chapter 7

Discussion

In this chapter, we list the limitations of our solution, and provide some future work in order to improve it.

7.1 Limitations

A first limitation of our solution is related to the component *Activation* (Context and Feature), and more specifically to the task *Enforce Dependencies*: in this task, we enforce the dependencies by firing any enabled internal transition in the Petri Net. As we said in the previous chapter (see Section 6.2.1), we might have *infinite loop* while executing this task, also known as *unstable steps* [4].

We illustrate this problem with the small Context Dependency Model of Figure 7.1 and its corresponding Petri Net in Figure 7.2: This is a variant of the model of our ERS application, where we have replaced the causality by an implication. The slight difference means that the `Map` is always active if we are `ConnectedAndLocation`.

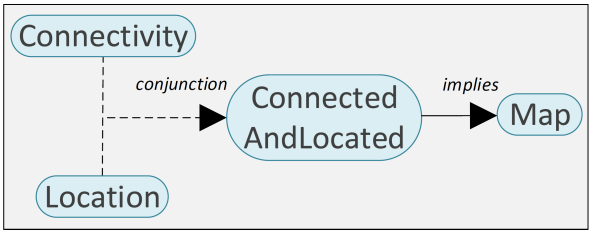


Figure 7.1 – Infinite Loops in `Activation` (Dependency Model).

We assume that all of these contexts are active: If we try to deactivate `Map` (by firing the external transition $req(\neg M)$), we will keep the same state of the Petri Net, as it is shown by the following sequence of enabled transitions:

$$req(\neg M) \rightarrow deac(M) \rightarrow deac_3(CL) \rightarrow deac(CL) \rightarrow deac_2(M) \rightarrow act(CL) \rightarrow act(M) \rightarrow * \; stop *$$

But we can also follow this sequence, and loop infinitely in the Petri Net model:

$$req(\neg M) \rightarrow deac(M) \rightarrow deac_3(CL) \rightarrow deac(CL) \rightarrow act(CL) \rightarrow act(M) \rightarrow deac(M) \rightarrow * \; loop *$$

It shows why we have to choose a random enabled internal transition in the Listing 6.2, with the instruction `its.sample.fire`.

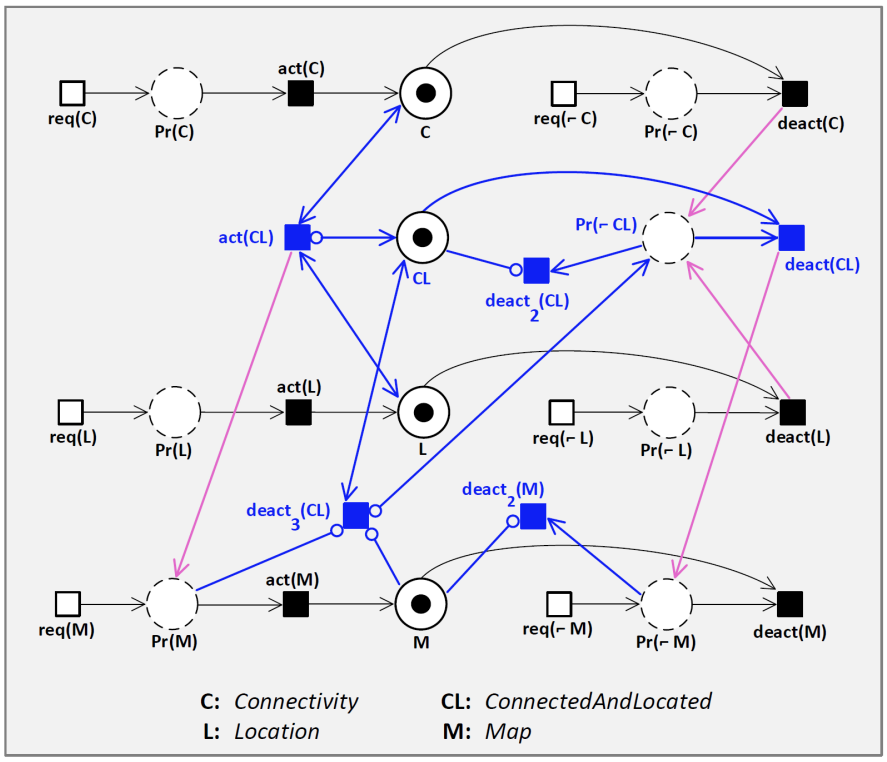


Figure 7.2 – Infinite Loops in `Activation` (CoPN).

A second limitation of our solution is that in some cases, we intentionally want to have conflicts, such as for the contexts `LowBattery` and `HighBattery` (`attempt_alternative` in Listing 6.3): the ERS application can activate `LowBattery` without deactivating `HighBattery` itself at the same time. The main reason of this design choice is that CoPN does not have any dependency relation that deactivates a context when another context becomes active, which might be useful to model a XOR relation in the Dependency Model.

A third limitation of our solution is related to alterations of type `:instance_attribute`, in which `value` is only set to existing objects

of the considered class (see Listing 6.8). Because of this, the developer of a context-aware application must also add an alteration of type `:instance_method` on the method `initialize` in order to update the newly created objects with `value`. It can be bothersome to add two alterations to express a single piece of information!

7.2 Future Work

A first future work would be to solve the limitations that we have listed in the previous section (see Section 7.1).

A second future work would be to make our framework support concurrent applications. Figure 7.3 illustrates a first approach, when the framework has a single thread t_F and the application has also a single thread t_A. We must ensure that the behavior of the application is consistent at any time: for instance, if it calls the method `alert` just after it has activated the feature `alertEmergencies`, it must behave accordingly by warning the user of nearby emergencies.

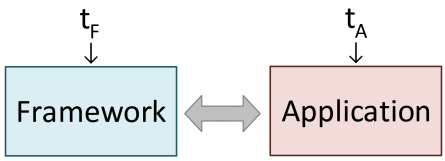


Figure 7.3 – Suggested Concurrency Model (One Thread per Component).

A third future work would be to improve the interactions with the framework for the developer of an application. We could develop a DSL, similarly to the code illustrated in Listing 7.1.

```
new_feature :alert_emergencies do
   mandatory_parent :alert_user
   requires :inform_emergencies
   alters :ERS do
      on_instance_method :alert do
         s = ""
         @emergencies.each do |e|
            s +=
               "#{e.type} detected nearby\n"
         end
         s
      end
   end
end
```

Listing 7.1 – Suggested DSL for Framework Interactions.

Chapter 8

Conclusion

This thesis had the intention to reconcile ideas from two software engineering domains, FOSD and COP, in order to facilitate the development of context-aware systems. At the completion of this work, this thesis has achieved the following contributions:

- **It has suggested a new formalism that reconciles ideas from FOSD and COP:** it makes use of *"activatable"* entities, for which *contexts* and *features* are specializations of this type of entity. It also introduces *"alterations"*, which are pieces of code that alter the base program. Contexts and features are modeled in isolation in their respective models (*Context Model* and *Feature Model*), and each of these models holds the inter-relations of the considered entities, in two sub-models (*Dependency Model* and *Configuration Model*). *Mappings* are links between particular contexts and features, where these contexts *enable* or *disable* features when they become active or inactive. It also proposes a *Conflict Resolution Policy* to resolve conflicts in the Context Model or the Feature Model;
- **It has provided an architecture and an implementation of a framework that rely on the above formalism**;
- **It has also tested the new modeling approach CoPN**, to model inter-dependencies between contexts and between features.

The thesis also contributed to model an ERS application with the notion of contexts and features, for which a prototype has been developed. This application has also been used to validate the our work.

Still, the framework shows some problems, as we discussed its limitations and missing aspects. In particular, we may improve the interactions with the framework, so that it is easier to develop applications.

We believe that this thesis initiates a merging of the results from FOSD and COP, which will enable the development of new kinds of applications that will be much more dynamic and smarter than today.

Bibliography

[1] Apel, S., Janda, F., Trujillo, S., and Kästner, C. Model superimposition in software product lines. In *International Conference on Theory and Practice of Model Transformations* (2009), Springer, pp. 4–19.

[2] Apel, S., Lengauer, C., Möller, B., and Kästner, C. An algebra for features and feature composition. In *International Conference on Algebraic Methodology and Software Technology* (2008), Springer, pp. 36–50.

[3] Batory, D. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines* (2005), Springer, pp. 7–20.

[4] Cardozo, N. *Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems*. PhD thesis, Université catholique de Louvain, 2013.

[5] Cardozo, N., De Meuter, W., Mens, K., González, S., and Orban, P.-Y. Features on demand. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (2014), ACM, p. 18.

[6] Cardozo, N., González, S., Mens, K., Van Der Straeten, R., Vallejos, J., and D'Hondt, T. Semantics for consistent activation in context-oriented systems. *Information and Software Technology 58* (2015), 71–94.

[7] Cardozo, N., Günther, S., D'Hondt, T., and Mens, K. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *Intl. Conf. on Software Engineering Advances. IARIA* (2011), Citeseer.

[8] Costanza, P., and Hirschfeld, R. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages* (2005), ACM, pp. 1–10.

[9] Dey, A. K. Understanding and using context. *Personal and ubiquitous computing 5*, 1 (2001), 4–7.

[10] Duhoux, B. L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte. Master's thesis, Université catholique de Louvain, 2016.

[11] Eshuis, R., and Dehnert, J. Reactive Petri nets for workflow modeling. In *International Conference on Application and Theory of Petri Nets* (2003), Springer, pp. 296–315.

[12] González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J., and Goffaux, J. Subjective-C: Bringing context to mobile platform programming. intl. conf. on software language engineering, 2011.

[13] González, S., Mens, K., Colacioiu, M., and Cazzola, W. Context traits: dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of the 12th annual international conference on Aspect-oriented software development* (2013), ACM, pp. 209–220.

[14] Günther, S., and Sunkle, S. rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming 77*, 3 (2012), 152–173.

[15] Hartmann, H., and Trew, T. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Software Product Line Conference, 2008. SPLC'08. 12th International* (2008), IEEE, pp. 12–21.

[16] Herrmann, S. Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance* (2010), ACM, p. 2.

[17] Hirschfeld, R., Costanza, P., and Nierstrasz, O. Context-oriented programming. *Journal of Object Technology 7*, 3 (2008).

[18] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., DTIC Document, 1990.

[19] Keays, R., and Rakotonirainy, A. Context-oriented programming. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access* (2003), ACM, pp. 9–16.

[20] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. *ECOOP'97—Object-oriented programming* (1997), 220–242.

[21] Krueger, C. W. New methods in software product line development. In *Software Product Line Conference, 2006 10th International* (2006), IEEE, pp. 95–99.

[22] Liu, J., Batory, D., and Lengauer, C. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 112–121.

[23] Mens, K. Programming Paradigms: theory, practice and applications. Master's lecture in Computer Science and Engineering, Université catholique de Louvain, 2016.

[24] Mens, K., Cardozo, N., and Duhoux, B. A Context-Oriented Software Architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming* (2016), ACM, pp. 7–12.

[25] Mens, K., Duhoux, B., and Cardozo, N. Managing the context interaction problem: a classification of conflict resolution techniques in dynamically adaptive software systems. In *Workshop on Live Adaptation of Software Systems. LASSY'17* (2017), ACM (soon published).

[26] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (1989), 541–580.

[27] Peterson, J. L. Petri net theory and the modeling of systems.

[28] Pohl, K., Böckle, G., and van Der Linden, F. J. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

[29] Poncelet, T., and Vigneron, L. The Phenomenal Gem. Master's thesis, Université catholique de Louvain, 2012.

[30] Prehofer, C. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience 13*, 6 (2001), 465–501.

[31] Rohn, E. Predicting context aware computing performance. *Ubiquity 2003*, February (2003), 1–17.

[32] Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines* (2010), Springer, pp. 77–91.

[33] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming 79* (2014), 70–85.

[34] Van Gurp, J., Bosch, J., and Svahnberg, M. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on* (2001), IEEE, pp. 45–54.

[35] Von Löwis, M., Denker, M., and Nierstrasz, O. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007* (2007), ACM, pp. 143–156.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**