

École polytechnique de Louvain

Towards a Testing Approach for Feature-Based Context-Oriented Programming Systems

Author: **Pierre MARTOU**
Supervisors: **Kim MENS, Benoît DUHOUX**
Reader: **Axel LEGAY**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Abstract

Over the last few years, research on systems able to adapt their behaviour to their environment attracted more and more attention. More specifically, feature-based context-oriented programming aims at creating a software that adapts to "contexts" by using concepts inspired from feature modelling. Developers who create systems with this paradigm need a way to test complex software. One problem is the exponential number of possible configurations, and thus, the impossibility to test them all. However, thanks to the testing approach we propose, inspired by combinatorial interaction testing (CIT), the number of test cases that needs to be considered is no longer exponential. By using the generated test cases, we can thoroughly test a feature-based context-oriented system with humanly readable scenarios.

Acknowledgements

I would like to thank my two supervisors, Prof. Kim Mens and Ph.D. student Benoît Duhoux, a lot. They both took the time to attend our "short" meetings on Teams, every Monday at 11:30 am, which more often than not lasted more than one hour. These discussions were always very inspiring, their advice always on point, and they guided me to where I am today. I thank them for their availability until the end.

I would also like to thank Prof. Axel Legay. He gave me pointers in the exploration phase of this master thesis, and kindly accepted to be a reader for it.

I also thank the people who reviewed my thesis, notably my parents, Elisabeth Kneip, and that random guy on Discord who said "Hey ! I can reread your thesis if you want" and found an error that went unnoticed.

Contents

Abstract	1
1 Introduction	4
2 Background	6
2.1 Feature modelling and related theories	6
2.1.1 Features	6
2.1.2 Feature modelling	7
2.1.3 Feature model constraints conversion	8
2.1.4 Software product lines	11
2.2 Context-aware systems	11
2.2.1 A variety of context-aware systems	11
2.2.2 Feature-based context-oriented systems	12
2.3 Testing studies in related fields	16
2.3.1 Automated testing	17
2.3.2 Interaction testing	17
2.3.3 SAT solving	19
2.4 Revision of the background	20
3 Case study	21
3.1 Contexts	21
3.2 Features	23
3.3 Mapping	24
3.4 Methodology	24
3.4.1 Iterative strategy	26
3.4.2 Iteration example	26
4 Objectives and challenges to testing feature-based context-oriented systems	29
4.1 Towards a testing methodology	29
4.2 Explosion of the number of interactions	30
4.3 Testing a single feature	31

4.4	Summary	32
5	Approach	33
5.1	Construction of the solution	33
5.1.1	CIT approach	33
5.1.2	Adaptation to a CIT approach	34
5.1.3	Find a suitable CIT approach	35
5.1.4	Adaptation to SAT solvers	37
5.1.5	Greedy CIT-SAT algorithm in theory	37
5.2	Summary	39
6	Implementation and results of the testing methodology	40
6.1	Testing methodology	40
6.1.1	Data extraction	41
6.1.2	Data parsing	43
6.1.3	SAT solving	43
6.1.4	Greedy CIT-SAT algorithm implementation	44
6.1.5	Result refining	44
6.2	Result	46
7	Validity	49
7.1	Practicality	49
7.2	Rapidity	50
7.3	Efficiency	50
7.4	Incrementality	50
7.5	Threats to validity	51
7.6	Limitations	51
8	Future work	53
8.1	Better interface	53
8.2	Optimization	54
8.3	Maintenance	54
8.4	Complete the testing methodology	55
8.5	Summary	55
9	Conclusion	56
	Bibliography	57
	Appendix	61

Chapter 1

Introduction

"Context-oriented systems" or "context-oriented programming" (COP) are a class of software that reacts to the environment and changes their inner workings during their execution. These kinds of systems have a lot of possible applications in different fields, which prompted several research teams to explore different approaches and gave birth to different points of view on these systems.

One of these is "feature-based context-oriented programming", which is a paradigm explored by Pr. Kim Mens and his RELEASEd research team. It combines notions from COP and feature modelling, amongst others. Different people have contributed to building the paradigm with previous master theses ([1], [2], [3], [4], [5]) and it is still under active development, notably thanks to the work of Benoît Duhoux as part of his PhD.

Context-oriented systems, by nature, have to adapt their behaviour to a changing environment at runtime. This implies that the software behind them can be very complex, with an exponential number of possible interactions within the system, an equally exponential number of possible scenarios, a very intricate architecture, etc. Previous work mainly focused on building the paradigm (architecture, methodology, tools) and it grew into a mature language prototype or framework, RubyCOP. That framework is built on top of Ruby to provide an easy way for developers to build feature-based context-oriented systems. It is also used to explore and validate novel contributions to this new paradigm of feature-base context-oriented programming.

Testing software systems with such intricate dynamic complexity can be challenging. However, testing is an important and non-optional step before a software system can be effectively used. Among possible applications, critical applications exist (for example, a previously explored application is a risk information system [6])

and such critical application cannot be released without robustness and effectiveness assurances.

Therefore, the goal of this master thesis is to contribute to the research on feature-based context-oriented systems, through something that all software in this world needs: testing. How to thoroughly test it raises several questions, and this master thesis tries to answer one of them: reducing the potentially exponential number of test cases that exist. To address this problem, we use techniques from combinatorial interaction testing (CIT) adapted to answer the needs of our particular class of software system.

A comprehensive conception methodology is currently explored by RELEASED. Whereas the main focus of this master thesis is to propose a novel testing methodology dedicated to feature-based context-oriented systems, we also contribute to exploring the conception methodology by describing the conception procedure we used to create our case study.

This master thesis is organized as follows: we first explain all concepts needed to understand the master thesis in Chapter 2. Then, a case study is proposed in Chapter 3 that we will use to produce and validate results. An analysis of the concrete objectives and challenges of this thesis follows in Chapter 4. Chapter 5 presents a theoretical account of what our solution consists of. An implementation of the proposed solution is shown in Chapter 6, which includes its cooperation with the existing RubyCOP language prototype. The usefulness, strengths and limitations of this approach are detailed in Chapter 7. What is left to do and what could be improved is described in Chapter 8. Finally, Chapter 9 concludes this master thesis by revisiting the main ideas, contributions and the future work.

Chapter 2

Background

Different key concepts or paradigms are explored throughout this thesis. This chapter is a background investigation, where each of these concepts are explained. We focus on the related work in three different fields: feature-oriented software development, context-aware systems, and testing techniques.

2.1 Feature modelling and related theories

2.1.1 Features

First of all, we should define what is a feature, a notion coming from feature-oriented software development (FOSD). However, FOSD was explored in many directions and by many research teams, and there is no definition that suits them all. The crudest definition would be that features are the building blocks of a software system. A previous survey of FOSD [7] sorted existing definitions from most abstract to most technical. The most abstract says that features are “*a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*” [8]. At the other end of the spectrum, we can say that they are “*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*” [9].

Consider a messaging application. One of its main features would be to send messages to another person. Another would be to add friends. A third more elaborate feature would be to sort one’s friend list according to some criteria. In this example, the first two features are the core of the application, which means they are mandatory, and they are called *commonalities*. The third feature is unnecessary to run the application, and there could be instances, variants or configurations

of the messaging application which do not offer this functionality. This kind of feature is called a *variability*.

2.1.2 Feature modelling

In order to model these features, we use Feature Diagrams. Two elements compose the model: the set of features and the relationships between them. We can visualize it as a tree structure. A minimalist example is shown in Figure 2.1, and a complete case study is presented in Chapter 3.

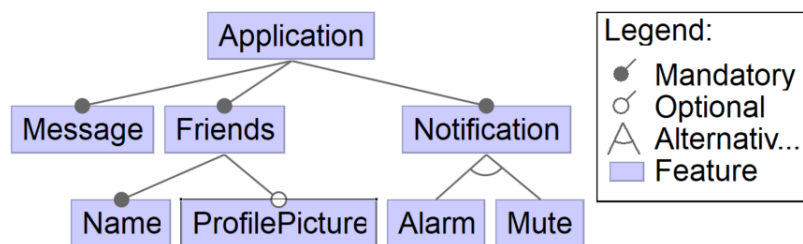


Figure 2.1: Minimalist feature diagram

To illustrate this notation, we will once again use a messaging application. The *commonalities* here are the *Message*, *Friends*, *Name* and *Notification*. The black filled circle above the *commonality* features represent a *mandatory* relationship with the feature above, also called the parent feature. Thus, the four of them share (indirectly for the *Name* feature) a mandatory relationship with the root node, effectively representing their *commonality* status. Conversely, we can find the *commonalities* of a feature diagram by retrieving all features directly or indirectly linked to the root node through mandatory relationships.

The *variability* lies with the *ProfilePicture*, *Alarm* and *Mute* features. *ProfilePicture* simply represents that you can improve someone's profile by attaching a picture to it. However, the application can run without it; it is an optional feature. It is represented by an empty circle above the feature's name.

When you receive a message, your communication device either alerts you with an *Alarm* or stays *Mute*. These two features share an *alternative* relationship. It means that only one of them can be selected at a time, but that at least one of them must be active if their parent feature is active.

We can define configurations from a Feature Model. A configuration is a subset of features said to be selected. A configuration is valid if the selected subset of features does not violate any constraints. For example, in our feature diagram, a configuration containing every feature is not valid, as *Mute* and *Alarm* cannot be selected at the same time. A valid one consists of all the features but *Mute*.

On a side note, the tool used to create the feature diagrams in this master thesis is FeatureIDE. It is an Eclipse-based IDE used in FOSD, more particularly in software product lines (see 2.1.4). It is under continuous development since 2004. For more details, please refer to <https://featureide.github.io/>.

2.1.3 Feature model constraints conversion

Each feature can be expressed as a Boolean variable which is True if this feature is selected and False if it is not. With this arrangement, there are as much Boolean variables as there are features. A feature model's configuration is equivalent to an assignment of truth values to every Boolean variable.

By considering the previous the example again, we will transcribe the feature *Mute* as the Boolean variable M_v and *Alarm* as A_v . When the first feature is selected and the second is not, in a configuration, M_v is True and A_v is False. One valid configuration would be an assignment of values such as shown in Table 2.1.

Feature	Message	Friends	Name	ProfilePicture	Notification	Alarm	Mute
Variable	M_v	F_v	Na_v	P_v	N_v	A_v	M_v
Value	True	True	True	False	True	False	True

Table 2.1: Simple configuration where ProfilePicture and Mute are disabled

With this arrangement, we can convert all of the feature model's constraints to propositional logic. Let $alternative(N_v, A_v, M_v)$ represent the alternative relationship between A_v and M_v with N_v as their parent feature. Its evaluation is equal to True when the relationship is satisfied by the configuration, and False if it violates it. In propositional logic, this is equivalent to:

$$alternative(N_v, A_v, M_v) \iff ((N_v \Leftrightarrow A_v \vee M_v) \wedge (A_v \Leftrightarrow \neg M_v \wedge N_v) \wedge (M_v \Leftrightarrow \neg A_v \wedge N_v))$$

Using propositional logic, we can transform all the constraints in similar Boolean expressions. A configuration, seen as a Boolean variable assignment, can thus be validated by checking that each expression remains True.

However, by taking a step further, we can convert them to conjunctive normal form (CNF). It allows us to use existing algorithms that only work for this kind of Boolean expressions, notably SAT solvers (see Section 2.3.3 for more details).

CNF formulas are a conjunction of clauses, where each clause is a disjunction of literals, and a literal is a variable (or its negation). The formula $(a \vee -b) \wedge (-a \vee b)$ is a CNF formula over the literals a and b (or their negation) with two clauses.

The converted formulas were largely inspired by the work of a previous master thesis: *"Consistency management of contexts and features in context-oriented programming language with SAT solving"* by Alexis Van den Bogaert [5]. Using CNF formulas with feature models constraints was also explored in previous works, such as *"SAT-based Analysis of Feature Models is Easy"* [10], in the context of Software Product Lines (see Section 2.1.4).

Table 2.2 shows all the basic constraints, their representation in feature diagram notation, and their formula converted into CNF. P stands for Parent Feature and C_n for the n th Child Feature, as there could be many Child Features.

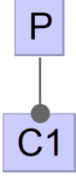
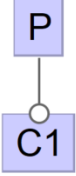
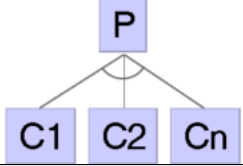
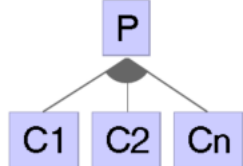
Constraint	Explanation	
	Representation	CNF formula
Mandatory	If P is not active, neither is C1. If P is active, C1 must be active.	
		$(P \vee \neg C_1) \wedge (\neg P \vee C_1)$
Optional	If P is not active, neither is C1. If P is active, C1 can be active or not.	
		$(\neg C_1 \vee P)$
Alternative	If P is not active, neither are any Child Feature. If P is active, one and only Child feature is also active.	
		$(\neg P \vee C_1 \vee C_2 \vee \dots \vee C_n) \wedge$ $(\wedge_{a=1}^n (\neg C_a \vee P)) \wedge$ $\wedge_{a=1}^n \wedge_{b=a+1}^n (\neg C_a \vee \neg C_b)$
Or	If P is not active, neither are any Child Feature. If P is active, at least one Child feature is also active.	
		$(\neg P \vee C_1 \vee C_2 \vee \dots \vee C_n) \wedge$ $(\wedge_{a=1}^n (\neg C_a \vee P))$

Table 2.2: Basic constraints of feature models, their representation, and their CNF formulas [5]

2.1.4 Software product lines

A Software Product Line (SPL) is a class of software systems representing an entire family of software systems instead of a single one. This variability is represented by a feature model: each configuration of this model represents a different software system. Each system of the family is said to possess a set of *assets*, the equivalent of *features* in FOSD.

SPL is a close paradigm to FOSD and, in a way, to context-aware systems. They all seek to create a multitude of slightly different software systems efficiently. This paradigm is important and popular, and benefits from a large literature waiting to be adapted to other domains. In particular, its literature was a source of inspiration to people exploring context-aware systems [11], and many articles played an important role in building context-oriented programming paradigms ([12], [13]).

2.2 Context-aware systems

2.2.1 A variety of context-aware systems

Context-aware systems modify their behaviour to adapt to their environment. A context can be seen as an entity that reifies a particularity of the environment. Context could be the brightness level in a room, or the temperature outside. However, just as with the definition of a feature, no definition of context-aware systems can accommodate all possible approaches in this domain. Instead, we can try to differentiate them.

In 2007, Baldauf differentiated context-aware systems by distinguishing the provenance of their context and their context management models [14]. Context can be collected either from a direct sensor access, from a middleware infrastructure hiding low-level sensors, or from a context server, a remote data source [15]. In this thesis, we do not focus on context acquisition, as we take it for granted in our experiments. We will simulate context changes manually.

Once context-aware systems collect data about their environment, they need to manage it. To this end, they can use widgets, networked services, or a blackboard model [16]. As time passed, more models emerged, and the feature modelling that we saw in 2.1.2 could be considered as one of them.

More recently, in 2016, Alegre defined another characteristic to differentiate context-aware systems [17]. According to him, context-aware systems are differentiated by their behaviour and their adaptation. Such a system can show an active execution (Self-Adaptivity), where the system acts by itself upon variation in the environment. Conversely, it could have a passive execution (Control) where the user keeps control.

In the same spirit, a context-aware system can either have an active configuration (Learning & Adapting), which automatically detects changes in the context with no need for a human to interfere. Otherwise, it could have a passive configuration (End-user programming), where the user provides himself data about his needs.

However, no matter the provenance of the context and the way to manage it, all context-aware systems share common motivations. They aim at improving the reuse of software systems that share a core set of capabilities for several different situations, but have to be adapted to each. For example, one user could be retired while another is a young child, but both use their smartphone to chat with friends. The messaging application they use has a lot in common, and can be adapted to better answer the needs of its users.

Most traditional programming paradigms suffer from software rigidity, a lack of modularity, and are made with the mindset that their environment is fixed. More than just reuse, context-aware systems integrate a deeper awareness of context, which allows them to implement innovative features impossible to implement with traditional systems. A smartphone application can perform better if it knows the device it runs on, the user's location, or the quality of its connectivity to the network.

With these motivations in mind, several variants of context-oriented programming (COP) have been proposed to ease the design and implementation of such systems.

2.2.2 Feature-based context-oriented systems

Feature-based context-oriented programming was first introduced by Duhoux et al. in 2016 [1, 18]. As its name implies, this paradigm incorporates features as first-class entities in its architecture, which distinguishes it from most traditional context-oriented programming (COP) languages. This architecture combines concepts from Feature-Oriented Software Development (FOSD) and Context-Oriented Programming.

The representation of an application's behavioural variability comes from FOSD, and more specifically feature modelling. A *feature model* represents all possible behaviours the system may have and the (de)activation of each feature corresponds to a change in behaviour. Feature-base context-oriented programming also uses feature modelling to imagine a *context model* that represents and manages all possible contexts (the environment to which the system may adapt), resulting in a clear separation between the notions of feature and context.

A variation in context is reflected by a (de)activation of contexts in the context model, which in turn (de)activates features in the feature model. This (de)activation of features adapts the behaviour of the system to its current context at runtime. After the feature and context models, the third element that governs feature-based context-oriented systems is the *mapping* between the two models. The architecture's control flow, which we explain below, is closely related to this operation (Figure 2.3).

The idea of a mapping between a Context Variability Model and a Feature Model was first explored by Hartmann in 2008 [12], as part of research on Software Product Lines. This modelling approach is very close to the one we use in feature-based context-oriented programming. Figure 2.2 is extracted from a paper by Hartmann and Trew, which proposed a way to model Multiple Product Lines (MPL) within a single system.

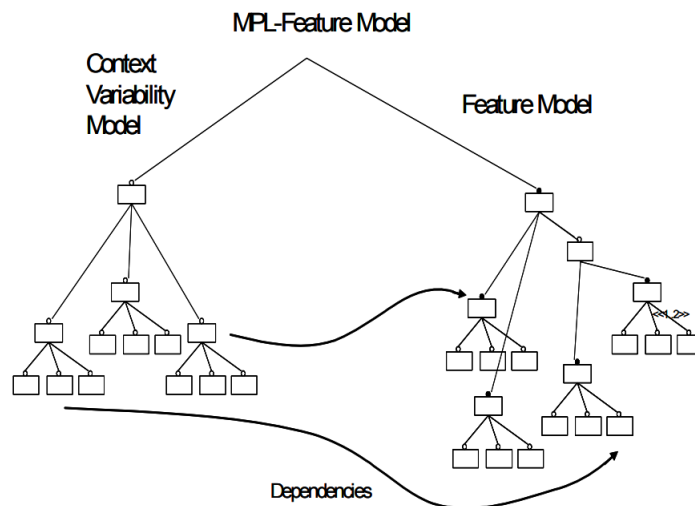


Figure 2.2: MPL-Feature Model Diagram, extracted from "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains" by Hartmann and Trew [12]

This modelling of contexts and features clearly separated two models and connected them through a formal mapping between them. This is the second main difference between feature-based context-oriented programming from other COP languages.

In this particular thesis, we will make use of a feature-based context-oriented programming language framework built on top of Ruby: RubyCOP. The framework's control flow consists of four major steps, as shown in Figure 2.3. These steps correspond to our earlier explanation regarding the operation of a feature-based context-oriented system.

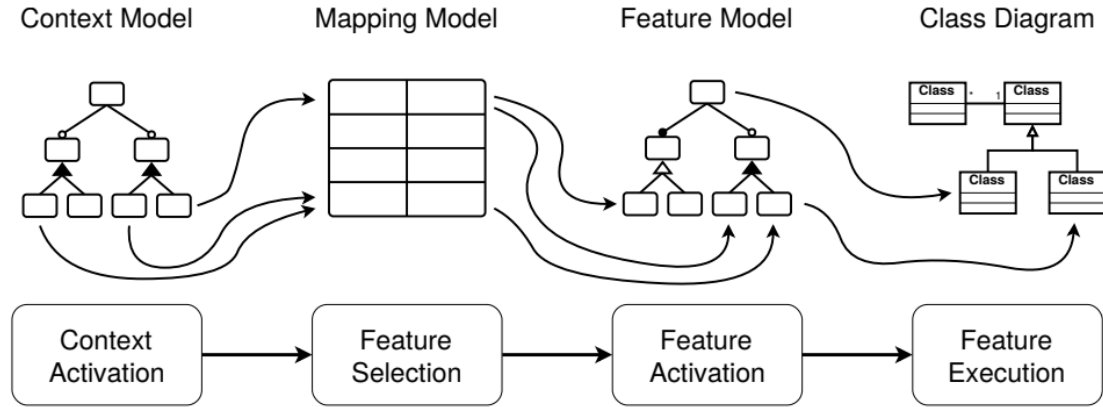


Figure 2.3: Feature-based context-oriented programming language control flow, extracted from "Implementation of a Feature-Based Context-Oriented Programming Language" [6]

When a context changes, the *Context Activation* component receives a message, informing it of this new discovery. It will update the context model's state accordingly by (de)activating the adequate context, which will succeed only if the final configuration satisfies the model's constraints. If not, the system rolls back to a previous valid configuration.

Once the change in context has been taken into account, the change will be reflected upon the features. The *Feature Selection* component's role is to select the features that should be (de)activated correspondingly to the changes in the context, based on the declared mapping between contexts and features.

The *Feature Activation* component works similar to its counterpart *Context Activation*. The component will try to (de)activate a set of selected features, but it

will only succeed if it does not conflict with the constraints imposed by the feature model. Again, if it fails, the system rolls back to a previous valid configuration.

Finally, the *Feature Execution* component will adapt the behaviour of the running system by deploying or removing the features' classes, according to the features' (de)activation.

The complete architecture consists of three parts. We already detailed the Architecture part, which is the framework's control flow. One of the other parts, Modelling, is the feature modelling part. It contains all the classes needed to code and represent feature and context models.

The third part, Entities, contains the main entities that programmers need to declare and define, such as the contexts, the features and the mapping between them.

The testing methodology developed in this thesis is built on top of this language and barely needs to modify it, as we will see in Chapter 6.1.1 on the data extraction. The part of the implementation architecture that concerns us the most is the Entities part and its classes, which we will concisely describe here.

Contexts and features are reified almost the same way in this architecture. Hence, we will present both of them in parallel.

The *Context* and *Feature* classes correspond to the notion of context and feature respectively, and each object of these classes represents a specific entity in their respective model. They inherit from an *Abstract* counterpart. *AbstractContext* or *AbstractFeature* are mostly infrastructural entities that help to structure the feature or context tree hierarchy. They are attached to no targeted classes in the application and do not alter it. These classes themselves inherit from *Entity*, which captures the commonality between features and contexts, namely that they can be modelled as a feature diagram. This common root Entity is logical, since both context and feature are modelled through the same kind of dependencies and constraints.

Their *Declaration* counterpart, *ContextDeclaration* and *FeatureDeclaration*, are classes that the programmer extends to declare his context and feature model. The declaration of a model starts by the definition of a root, which is necessarily abstract. Just like *Context* and *Feature* inherit of *Entity*, their *Declaration* counterpart inherits from *EntityDeclaration*.

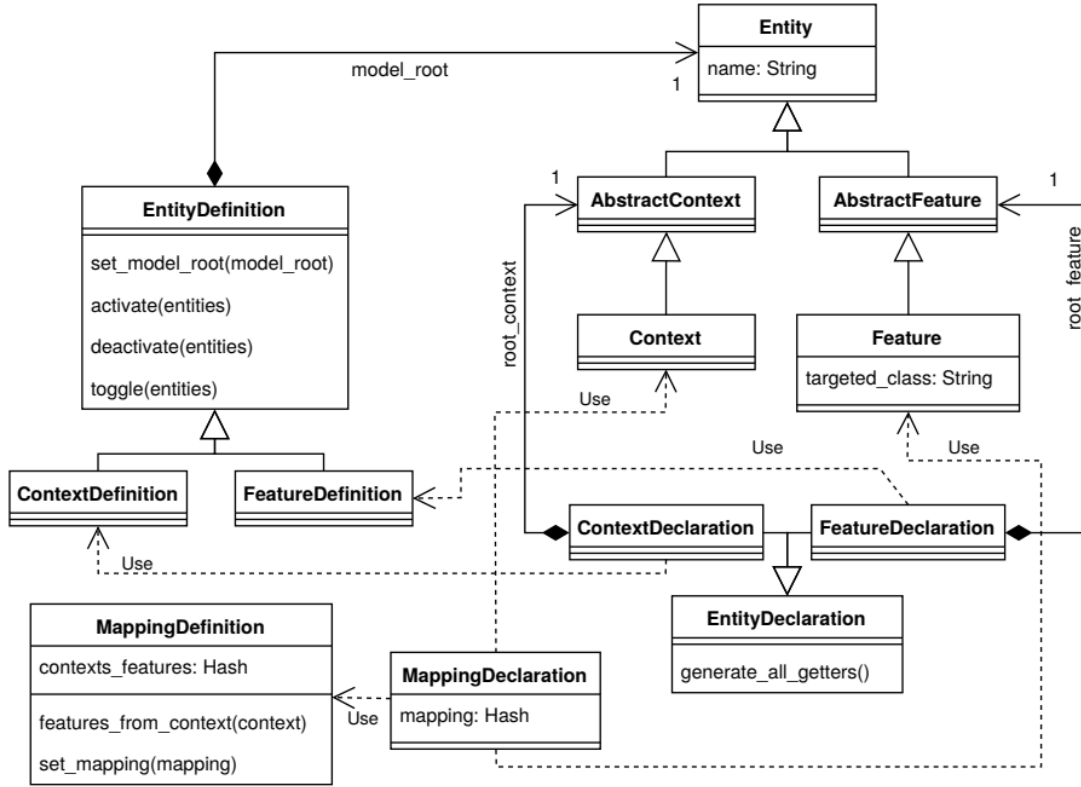


Figure 2.4: Zoom on the Entities part of the implementation architecture, extracted from "Implementation of a Feature-Based Context-Oriented Programming Language" [6]

The *Definition* counterpart's role is to rule over the activation and deactivation of the feature/context they represent. They are created from the context and feature declarations provided in the *Declaration* counterpart.

Finally, in the same line of thinking, *MappingDeclaration* is a class used by the application programmer to define a mapping between contexts and features. *MappingDefinition* is the class used at runtime to effectively interpret this mapping declaration.

2.3 Testing studies in related fields

We will begin this section by introducing test automation in all generality. We will then focus on a specific topic, combinatorial interaction testing. Finally, we will

address the subject of SAT solving, a powerful tool often used in automatic testing.

2.3.1 Automated testing

Automated software testing means the automation of software testing activities usually conducted by humans. Test automation is conducted by constructing software tools a layer above the software to test. The usual goal is cost savings in the field of testing, higher software quality, or to allow continuous delivery and continuous testing for fast-growing applications [19].

Another motivation for automated testing is to handle testing a human cannot achieve. Highly-configurable systems have been in existence for a long time now, systems for which testing all configurations is nearly impossible. To address this problem, sampling techniques such as combinatorial interaction testing have been proposed [20], [21]. They usually use a feature model to manage the variability in such systems and apply a specific coverage criterion to generate a decently-sized set of test cases, optimized by the coverage criterion to answer to a specific need. A coverage criterion could be pairwise feature interaction or dissimilarity [22].

This is the case of SPL. The literature proposes multiple ways to handle this problem, such as multi-objective test generation [23], or more recently many-objective generation [24]. Sampling techniques refinements for SPL have also been researched, such as statistical prioritization [22].

As combinatorial interaction testing inspired us the most in this master thesis, we present its notions in more detail in the following section.

2.3.2 Interaction testing

Combinatorial interaction testing (CIT) was researched very early on, already in the nineties, and has a long history [20]. The set of test cases generated by CIT is called the CIT sample. Its generation is based on the principle of *covering arrays*.

Let us for example consider a set of configurable features for a smartphone: *Brand*, *State* and *Performance*. These features are an abstract notion, but they can take multiple values called variants. Our example's variants are shown in Table 2.3.

A pairwise or two-way CIT sample combines all pairs of variants between any two of the features. For example, a Broken smartphone will be tested at least once with the Samsung variant, the Nokia variant, the Slow variant and the Fast variant.

Features	Variants
Brand	Nokia, Samsung
State	Broken, Used, New
Performance	Slow, Fast

Table 2.3: Features illustrating combinatorial interaction testing and their variants

This sample is also called *covering array*. This concept can be generalized to *t-way covering array*, where t is its strength. An example of a two-way covering array for our example is shown in Table 2.4.

Test case number	Brand	State	Performance
1	Nokia	Used	Slow
2	Nokia	Broken	Fast
3	Samsung	Used	Fast
4	Samsung	Broken	Slow
5	Samsung	New	Slow
6	Nokia	New	Fast

Table 2.4: Example of a two-way covering array

The idea of using a t -way covering array when testing a configurable system is to limit ourselves to the combinations of feature variants proposed by this covering array rather than testing all possible combinations of these variants, while still testing every t -way interaction possible.

Discovering a covering array is thus the main goal of combinatorial interaction testing. For now, no lower bound on covering arrays' size has been discovered to define what would the "best" covering array [25], so discovering the smallest one is a secondary goal.

An interesting property of these covering arrays is that their size grows logarithmically in the number of variants [20]. That makes them particularly attractive to highly-configurable systems, which can have a very large number of variants.

To discover a covering array, the most popular way is exploring the search space containing all possible configurations. For example, we can use a metaheuristic

search such as simulated annealing, genetic algorithms, or tabu search [26] [27].

Another search method is the greedy approach. There are two main strategies: the In-Parameter-Order (IPO) approach [28] and the one-row-at-a-time strategy [20][21]. The principle of the one-row-at-a-time is as its name implies: it will construct one test case, one "row", at a time, and add it to a growing array. Each time it constructs a new test case, it tries to create the best one according to some criteria, like the number of variant pairs this new test case will cover once it is added to the growing array. The algorithm stops when this array has grown into a covering array by covering all possible variant pairs.

We decided to use a greedy approach in our solution. More details about it will be given in Chapter 5. Moreover, 2-way covering arrays are often enough to test a system [21], and the size of a t-way covering array grows exponentially with t [20]. For these reasons, we will only consider 2-way covering arrays for the rest of this master thesis.

2.3.3 SAT solving

The final section of this chapter is on Boolean satisfiability (SAT) solvers. We will first define what is the Boolean SAT problem, then why we will use SAT solvers.

Let us consider a given Boolean formula. The SAT problem is determining if the variables in this formula can be consistently assigned to True or False in such a way that the formula evaluates to True. If an interpretation can be found which satisfies this Boolean formula, the formula is called *satisfiable*. If no such assignment exists, the formula is *unsatisfiable*. For example, $(a \wedge \neg b)$ is satisfiable with the assignment $a = \text{True}$ and $b = \text{False}$.

SAT solvers are tools whose goal is to handle this problem, and most current ones require the user to provide Boolean expressions in Conjunctive Normal Form (CNF). These solvers have been under active development for years. Nowadays, state-of-the-art SAT solvers are powerful tools, thanks to several advances in their technology [29] [30] [31]. They are able to solve problems with millions of constraints and hundreds of thousands of variables [32].

They prove that a Boolean expression is satisfiable by finding an appropriate assignment. Some approaches exploit this property and use SAT solvers to find a possible assignment to a satisfiable Boolean expression.

SAT solvers are thus an ideal choice to test Boolean expression. As we have seen in 2.1.3, the constraints defined in a feature diagram can be converted in CNF Boolean expressions. Moreover, previous studies showed that using SAT solvers to handle the specific constraints of a feature diagram is efficient [10]. Consequently, in this master thesis, we will use such a SAT solver to handle the various constraints found in a feature-based context-oriented system, more specifically in its context and feature model and the mapping between them (more details in Chapter 5.1.4 on the adaptation of a feature-based context-oriented systems to SAT solvers, and Chapter 6.1.3 on the actual implementation of a SAT solver).

2.4 Revision of the background

We started by discussing feature modelling. After a general introduction of context-aware systems, that knowledge was useful to understand and define feature-based context-oriented programming. We reviewed in detail its main concepts, its main differences with other COP languages, and its architecture, as the purpose of this master thesis is to develop a testing methodology tailored to this kind of system. To better understand the testing methodology we propose in this master thesis, we explored the field of automated testing, a vast domain that we briefly introduced. We elaborated on combinatorial interaction testing, which is central to the proposed testing methodology, as its main algorithm comes from CIT. We ended our journey by taking a look at SAT solvers, which will be helpful tools to handle the constraints of feature-based context-oriented systems.

Now that we have laid a solid foundation for our master thesis, we will continue on with our case study.

Chapter 3

Case study

To study feature-based context-oriented systems, we naturally need a concrete application on which to try out our ideas. It will be used whenever a practical case is needed. This chapter will both present the case study used in this master thesis and the way to conceive it. There was recently a methodology proposed by RELEASeD to build feature-based context-oriented systems, and we will provide in this chapter a complete analysis as to how this methodology was effectively used to conceive the case study.

The case study is a messaging application. Its main purpose is to send and receive messages, as well as to manage friends and groups. More features can be added to better handle particular context, for example to take into account the connectivity. As such, a messaging application is suitable to be conceived as a feature-based context-oriented system.

As any context-aware system, it can be greatly expanded to adapt to a lot more different situations. The same goes for the features, as we can add much more features. However, we try here to keep a system small enough to be able to quickly understand and experiment with it, and big enough to remain representative for the complexity of the problem at hand.

3.1 Contexts

The context model has four important parts, as illustrated in Figure 3.1. From top to bottom:

- *DeviceCapabilities* represents the user's device (smartphone or possibly more exotic devices) capabilities. It is optional, as the application can run in a

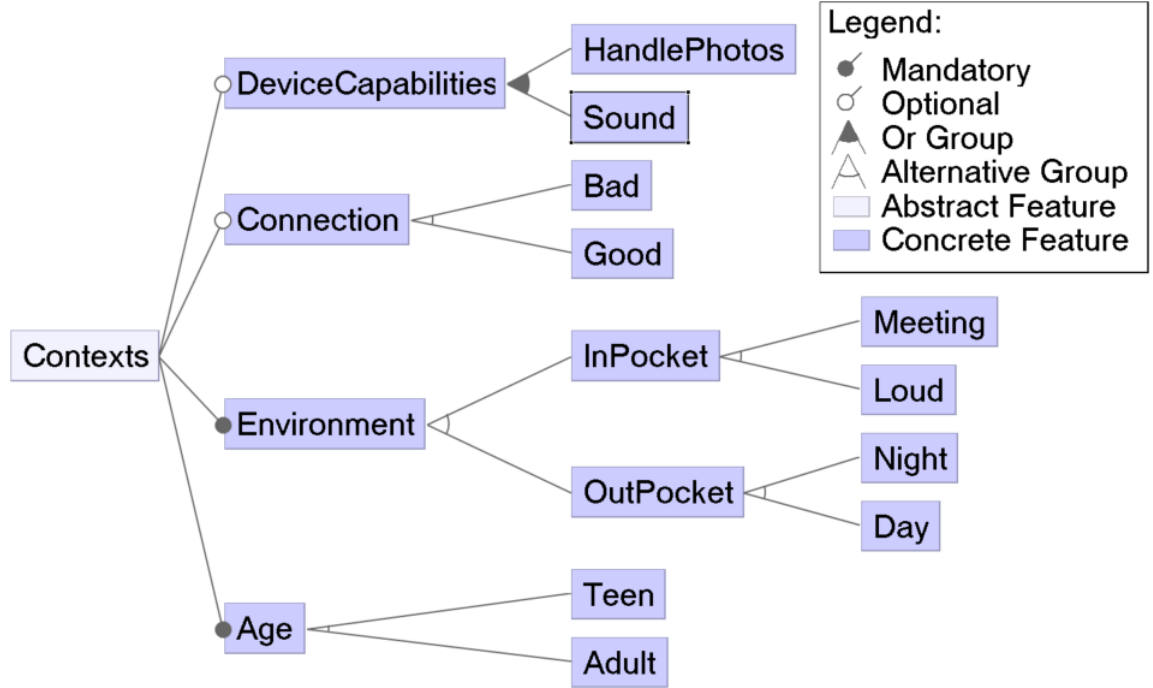


Figure 3.1: Context model

standard device-agnostic mode, but some features need to know whether the device can support photos (*HandlePhotos* context) or if it handles sound (*Sound* context). For simplicity, *HandlePhotos* represents both the device’s ability to send and to receive message sustaining photos. For example, a modern smartphone with a LED screen can handle photos. The same goes for *Sound*, which also represents the ability to record the voice on the device such as with a microphone.

- *Connection* represents the quality of the current Internet connection. It can be either *Bad* or *Good*, where a *Bad* connection means that it may have difficulty receiving some messages, such as photos.
- *Environment* is the situation the user and its device are in. We modelled it by separating the cases where the device is in the user’s pocket (*InPocket*) or not (*OutPocket*). When the device is in his pocket, the user can either be in a *Meeting*, for example at work, or in a *Loud* environment, for example in the street or while commuting. When it is not, we only consider if it is *Night*, for example when the user is sleeping, or *Day*, for example when the user left its device somewhere in his house or on his desk.

- *Age* is simply the age category of the user. He can either be a *Teen* or an *Adult*, as we stigmatised these two age groups to model two different ways of using a messaging application.

3.2 Features

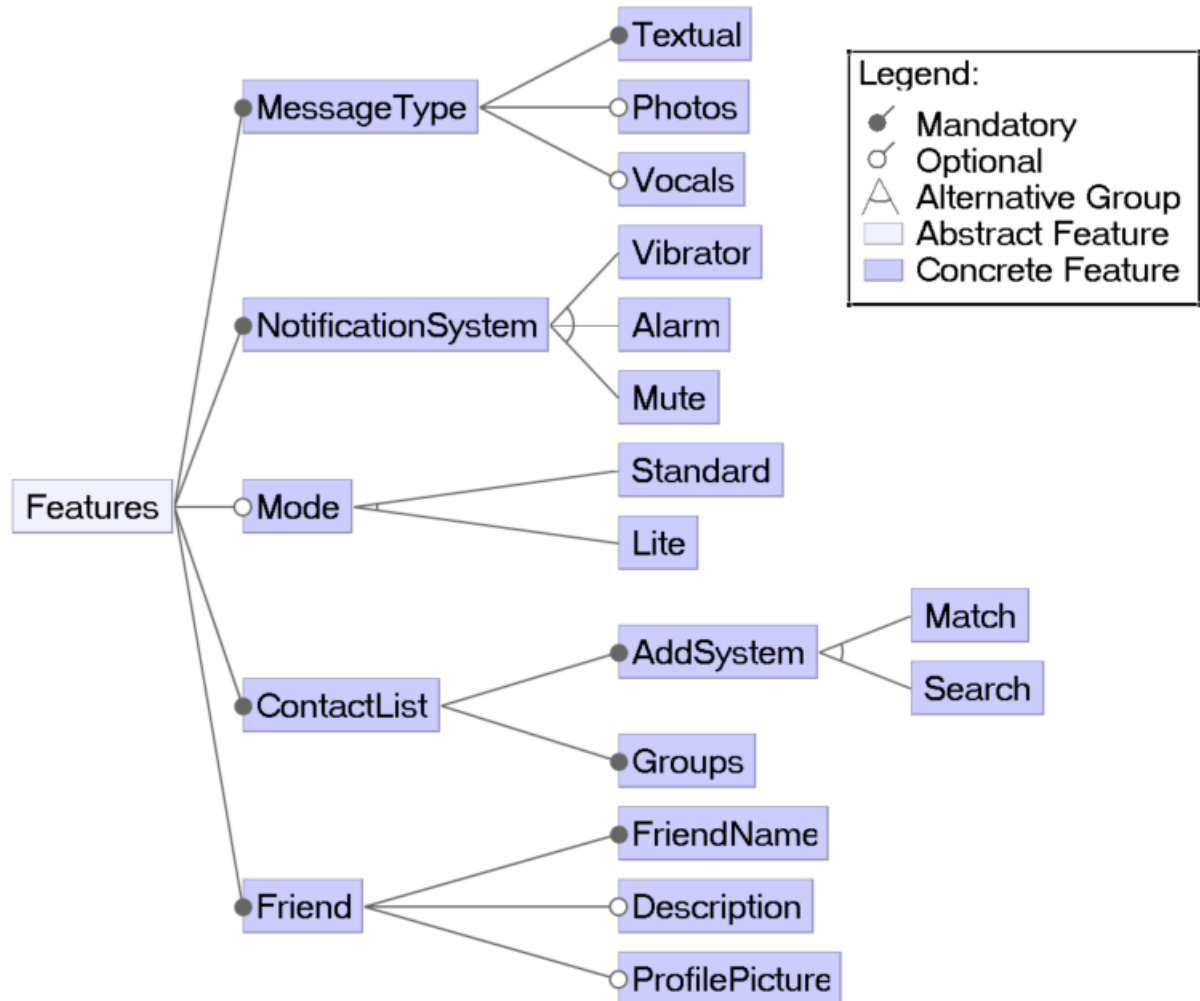


Figure 3.2: Feature model

The feature model has five important parts, as illustrated in Figure 3.2. From top to bottom:

- *MessageType* represents which type of message can be sent or received. The most basic one, a *Textual* message, is part of the core of the application and thus is mandatory. You could also send *Photos* (which also represents GIFs, or anything else than just text) or *Vocals*, which is an audio recording, but these are optional as they may not be available for all kinds of devices.
- The *Notification System* is what alerts you when you receive a message. You can either be alerted by a *Vibrator*, an *Alarm*, or the application can stay *Mute* if it decides it is better not to disturb you, for example because you are sleeping.
- *Mode* is either *Standard*, which does not limit the application in any way, or *Lite*, which handles the connection better by avoiding downloading photos, using a lighter version of the application, etc.
- *ContactList* is where you manage your friends. How you add them to your friend list is dependent on the *AddSystem*. You either use a matching system (*Match*) which presents you potential soon-to-be friends that you can add, or a simple searching system *Search* where you can search for your friends by name and add them. You can also create *Groups* with your friends to send messages to all of them at the same time.
- *Profile* serves as your profile. All profiles have a *FriendName* that defines them, and can possibly add a *Description* and a *ProfilePicture* to it.

3.3 Mapping

Table 3.1 presents the mapping of contexts to features for our case study. The contexts in the left column activate all the features in the right column. If there are multiple contexts, it means that both contexts should be activated in order for the features in the right column to be activated.

3.4 Methodology

Feature-based context-oriented systems can become complex as more and more contexts or features add up. The developer needs to think about what contexts its application could adapt to, with which features, and which contexts trigger which features.

Contexts	Features
Bad	Lite
Good	Standard
HandlePhotos, Good	Photos
Sound	Vocals
Meeting	Mute
Loud	Vibrator
Night	Mute
Day, Sound	Alarm
Teen	Match, ProfilePicture
Adult	Search, Description

Table 3.1: Contexts-to-features mapping of the case study

To address this modelling, the RELEASeD team proposed a particular methodology to conceive such systems, built around ideas that developers should keep in mind. The final objective is a complete methodology that goes from the initial requirements' analysis of the application until the testing of its implementation, as shown in Figure 3.3. It was extracted from the course "LINGI2252 – Software Maintenance and Evolution", lesson "10c - Feature Based Context Oriented Methodology", a course taught by Prof. Kim Mens, with course assistant Benoît Duhoux.



Figure 3.3: Steps of the researched methodology

The goal of this section is to apply and provide a feedback on this methodology on a practical case when applying the two first steps, the conception of the system. First, we will suggest a way to effectively use the methodology, then illustrate it on the example of this specific case study.

3.4.1 Iterative strategy

One of the main ideas is to keep always in mind both contexts and features together when conceiving the system. For each feature and context identified in the first step (Requirements), you should ask yourself *"In what context(s) is this feature relevant?"*, *"What other features are relevant in this context?"*, and *"What other contexts could affect this feature?"*.

We applied the methodology by constructing an iterative model in 4 steps. This model works by keeping a *"bag of ideas"* where ideas for future features and contexts are stored, and which should be initialized with one or two elements. It is schematized in Figure 3.4. These iteration steps are only there to guide the designer, and they can be followed strictly or not.

The goal of following these iteration steps is mainly to serve as a guideline to find more ideas, in order to boost a designer's creativity. It only needs one or two ideas to start iterating, and the following ideas come easily.

The second goal is to develop a clear architecture and model from the get-go. You can build your context model, your feature model and the mapping between them incrementally while iterating. For example, almost all ideas from step 2 will be *Sibling Features*, while matching contexts of a feature found in step 3 will easily be modelled in the mapping.

3.4.2 Iteration example

A concrete iteration is documented in Table 3.2, as it could help future developers to better understand the method and contribute to provide a feedback on the methodology.

This table also concludes Chapter 3. We defined a complete feature-based context-oriented system by detailing each part of its context model and its feature model, and by specifying the mapping between contexts and features. This case study will notably be used to try out the solution we propose in this master thesis and produce results.

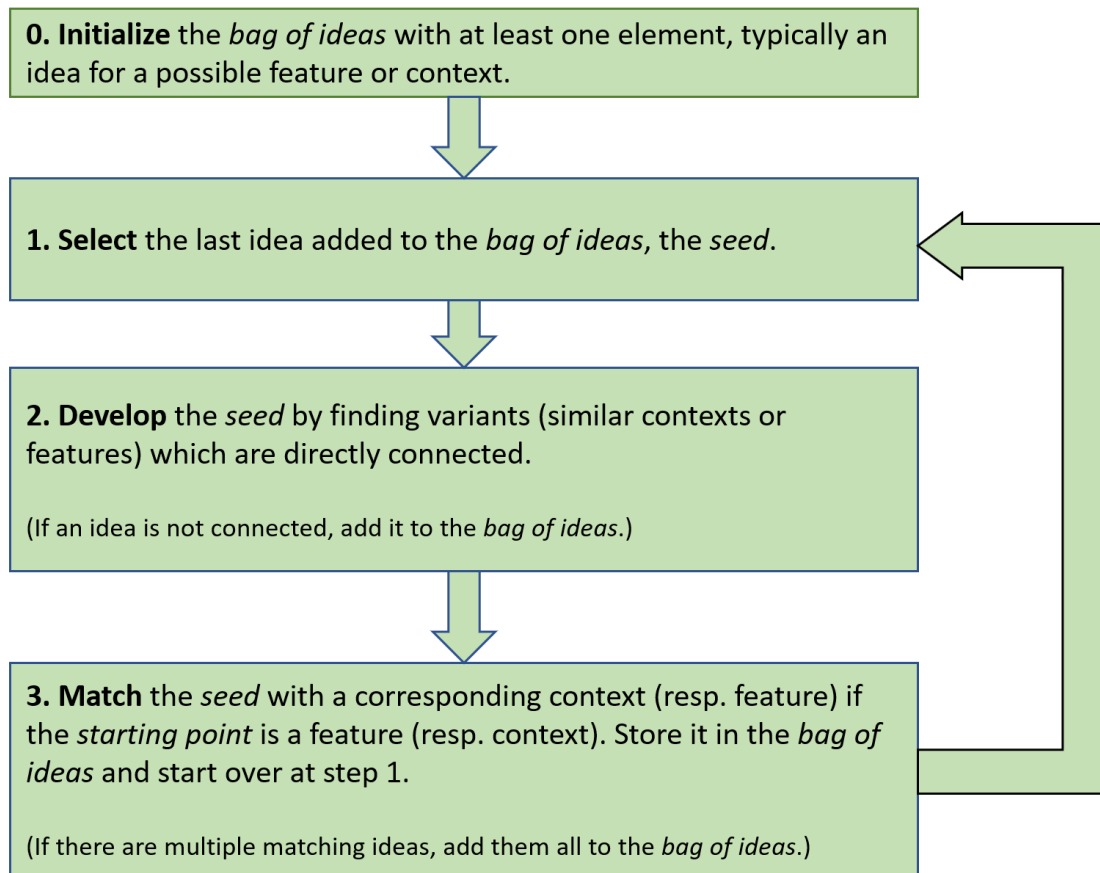


Figure 3.4: Iterative strategy

We also described our own iterative strategy, which is the way we applied the methodology proposed by the RELEASED team to conceive such a system. This strategy will serve as a feedback for future research on this topic.

Step	Element type	Thought process	Actions on the <i>bag of ideas</i>
0	Feature	Basic feature of a messaging application: MessageType	Add MessageType
1	Feature	MessageType is the <i>seed</i>	Remove MessageType
2	Feature	A message could be textual, a photo, or a recording.	Add Textual, Photos, Vocals
3	Context	To handle photos and such, you need the ability to do so.	Add DeviceCapabilities
1	Context	DeviceCapabilities is the new <i>seed</i>	Remove DeviceCapabilities
2	Context	What you can do with your smartphone is not only limited to its ability, but also to your connection	Add Connection
3	Feature	With speakers (from DeviceCapabilities), your smartphone can use an alarm.	Add NotificationSystem, Add Alarm
1	Feature	Alarm is the new <i>seed</i> .	Remove Alarm
2	Feature	You can be notified by an alarm, or by a vibrator, or not be notified at all.	Add Vibrator, Mute
3	Context	What are the circumstances where your smartphone should use the Mute mode,?	Add Environment, InPocket, OutPocket, ...
1	Feature	I have no more ideas concerning Environment and its Child Features, so I'll start over with Connection	Remove Connection
...			

Table 3.2: Iteration example

Chapter 4

Objectives and challenges to testing feature-based context-oriented systems

Now that we are on the same page for all the concepts and that we have a concrete application to validate our results, we will see exactly why we should test it and why it is non-trivial.

To this end, we will first discuss the need of a testing "methodology" before highlighting the specific challenges induced by the connections between the features' behaviour, notably the exponential number of possible test cases to be considered. Finally, we focus on the problems appearing when you want to test a single feature, which stems from the architecture of the framework itself.

4.1 Towards a testing methodology

First, we should remember what we are testing. We are testing a feature-based context-oriented system built on top of a Ruby framework. This language framework handles specific operations such as context and feature activation, etc.

A system that uses a complex framework and an important number of contexts and features is bound to become complex itself. The system's inherent complexity was already one of the motivations of previous work such as the development of a Feature Visualiser tool in RubyCOP [33] [34] [35].

The goal of RubyCOP is to support the development of a variety of such complex systems that are all based on the same concepts of contexts and features. Hence, it

seems logical to create a specific testing methodology common to these systems. The two following sections zoom in on two issues that stem from that complexity, and which are common to all feature-based context-oriented systems.

A developer that wants to use this programming framework for its application is not supposed to be aware of all its inner workings. Ideally, the framework should remain a black box to the developer. As a result, we need appropriate tools to help him use the framework and test his software without having to modify the framework and tinker with it.

Finally, we need to be sure of the quality, correctness and robustness of software written in this framework. Systems of all sorts can be written in the language framework, including critical applications such as a risk information system [6]. We could also easily imagine a context-aware system for hospitals to manage medical machines that adapts to the situation of the hospital and the patient, or a system that helps in driving a car according to the driver, specific car, or the road it is on. Such system absolutely cannot be released without thorough testing. Consequently, the best way is to propose an encompassing testing methodology and tool, instead of testing it blindly and partially.

4.2 Explosion of the number of interactions

The software systems we work with have a lot of different features and most of these are highly connected by nature, as they are part of a feature model. Moreover, the context model and the mapping add another layer of complexity to the possible interactions.

These connections could be the cause of numerous bugs. For example, suppose feature A always relies on an attribute from feature B, and feature B is active at the same time as feature A in all configurations but one. Or suppose that some feature C overrides another feature D involuntarily, but C and D are rarely in the same configuration. How can we know which configurations are to be tested?

The problem is that such undesired or unexpected interactions are not easily discovered. From the beginning, features are not meant to be totally independent parts of the system. It is hard to know whether you should test feature A with feature B or with feature C, as it is hard to determine whether A has an actual influence on B or C. Moreover, even if you partially know, it is hard to create scenarios for a subset of features, as the features' (de)activation is triggered by a

context model and mapping. All in all, manually creating a set of test cases with good coverage becomes nearly impossible.

Following this reasoning, we should just test all possible interactions and all features with each other, in other words: test all possible configurations. However, this approach would result in a combinatorial explosion of the number of test cases that should be considered. It is exponential in the number of features. It is simple to see: adding an *optional* feature to an existing feature model multiplies all possible configurations by two, or adding an *alternative* relationship with three children features multiplies all possible configurations by three.

How to reduce the explosive number of test cases in a manner that still guarantees testing a majority of interactions is the main issue this master thesis will focus on.

4.3 Testing a single feature

Finally, let us now consider the point of view of testing a single feature, which we know is faulty or is under development, or that is part of a configuration we are currently testing.

As mentioned previously, a feature is not independent. We cannot possibly deactivate all contexts and all features but one to test it: it would be (in most cases) an invalid configuration, according to the context and feature models. It is normal if an invalid configuration does not work: a *Mandatory* Child Feature is not supposed to work without its Parent Feature.

Beyond the independence problem, the mechanisms of the framework are problematic too. For example, RubyCOP provides a *proceed* mechanism to allow features to refine the behaviour of others [6]. In simple terms, by using it, we can refine the behaviour of a method with a feature without overriding all previous refinements to this method (i.e. by other features). It is a very powerful incremental modification mechanism to dynamically adapt a program's behaviour depending on the currently active contexts and features. Because of this *proceed* mechanism, it is hard to test a feature in isolation of the features it refines.

Another problem coming from the *proceed* mechanism is that the order of feature activation can have an impact. Imagine you activate the features A, then B, then C, all modifying a method *M*. When executing *M* while C is active, a *proceed* call may execute the changes made by A and B in an order which would be different

than if you had first activated B, then A, then C. When testing C, you do not want to see a behaviour changing randomly between runs.

As previously mentioned, the language framework should remain as close as possible to a black box. Hence, we can only solve those problems with a testing tool, as the developer alone cannot do it.

How to isolate the behaviour and errors of a single feature is the final issue discussed in this chapter, but will not be the main focus of this master thesis.

4.4 Summary

We argued that building a testing methodology was necessary for feature-based context-oriented systems. We also saw two issues coming from two different perspectives stopping us from building it. The first one concerned the combinatorial explosive of the number of possible test cases if you want to test them all. This issue was addressed in this master thesis, thanks to techniques from automatic testing.

The second issue is closer to an architectural problem, and was focused on the *proceed* mechanism. We will need innovative solutions, maybe by modifying the framework's design. This point is mentioned in Chapter 8 Future Work, as it is not the main focus of this master thesis.

Chapter 5

Approach

In order to reduce the exponential number of test cases to be considered, we defined a testing approach inspired by combinatorial interaction testing (CIT). This chapter will first explain step by step our choices in building a solution and what led us to them. We will end with a description of the final algorithm, which is at the core of the proposed testing methodology.

5.1 Construction of the solution

5.1.1 CIT approach

Our goal is to generate a decently-sized set of tractable, fault-finding test cases. To this end, we use a sampling technique coming from CIT to generate a covering array. Multiple previous highly-configurable systems chose this solution, and many studies have shown that these techniques efficiently find certain types of faults [21] [20] and may provide good code coverage [21] [20] [36].

Consequently, the core of the testing approach is an algorithm inspired by CIT whose goal is to produce a covering array for a feature-based context-oriented system.

However, CIT algorithms and covering arrays are made to handle features with a number of variants, such as illustrated in section 2.3.2, and not feature diagrams. Moreover, we cannot generate a covering array for the contexts and another one for the features. The problem is that configurations of contexts from the first covering array will, most of the time, be incompatible with the configurations of features from the second covering array. Indeed, they will not respect the mapping between contexts and features.

To solve these issues, we have to define an equivalent, alternative, representation to feature-based context-oriented systems, a kind of variable change which suits CIT algorithms.

5.1.2 Adaptation to a CIT approach

A test case, for a feature-based context-oriented system, is a specific set of (de)activated contexts and features. Therefore, our goal is generating a covering array whose variants' assignment is equivalent to such a set.

To this end, we unify contexts and features and represent them both in a single feature diagram whose root is connected to two *mandatory* child features: Contexts and Features. Below Contexts and Features are, respectively, the context model and the feature model of a feature-based context-oriented system. This new representation will be called the *unified* model, and the features that belong to this new feature diagram will be called the *unified* nodes. If we take the example of our case study, *Contexts* is a *unified* node just like the feature *Friends* or the context *Connection*. This new feature diagram is illustrated in Figure 5.1.

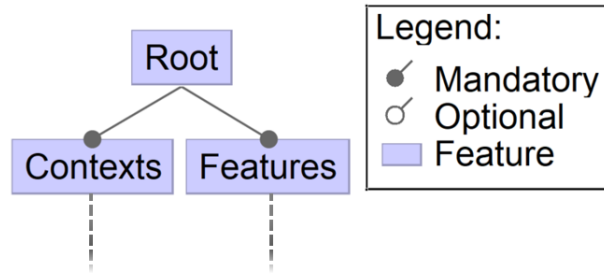


Figure 5.1: Representation of the unified feature diagram

To complete that unified model, we need to add the constraints from the mapping between contexts and features. Fortunately, doing so is not difficult: feature diagrams can support more complex constraints, including dependencies (requirement, exclusion, equivalence, ...). A mapping between context A and feature B corresponds to an equivalence dependency between A and B. They are either both activated, or both deactivated.

In summary, the final unified model is an alternative representation of feature-based context-oriented systems. It consists of a feature diagram where the context and feature models are fused with their roots as *mandatory* unified nodes below the

unified model's roots. Finally, this feature diagram supports a list of equivalence dependencies which corresponds to the mapping between contexts and features.

This unified model's nodes all have an equivalent feature or context, and vice versa. Likewise, this unified model's valid configurations are equally valid when applied to its corresponding feature-based context-oriented system, and vice versa. Thus, generating a covering array for a unified model is the same as generating one for its corresponding system !

The last step to adapt the unified model to CIT techniques is simply representing each unified node as a Boolean variable with value set to True if the node is activated, else False. We already saw this kind of conversion in section 2.1.3, where we expressed a feature diagram's configuration as an assignment of truth values to Boolean variables representing features.

In terms of a CIT problem, the features to be tested are the unified nodes. There are two variants to each node: True and False. A test case in the generated covering array is an arrangement of variants, which corresponds to an assignment of truth values, which itself corresponds to a set of (de)activated contexts and features. The generated covering array will effectively test all possible interactions between activated and deactivated nodes, and the contexts/features they represent.

5.1.3 Find a suitable CIT approach

To generate covering arrays, many different CIT sampling techniques exist. The main factor in deciding which one to use is the omnipresence of constraints in the modelling of a feature-based context-oriented system. We need a covering array that does not propose invalid configurations.

Traditional CIT sampling techniques could construct a complete covering array without taking these constraints into account. Then, they could delete the configurations in violation with these constraints and resume constructing their covering array in order to regenerate new configurations. They will cover the pairs lost with the deleted configurations.

However, let us take the example of our case study. There are 96 possible valid configurations in its context model, and 288 in its feature model. Without considering the mapping between the two models, there are 27648 valid configurations in the unified model, which is a largely overestimated number. Conversely, if constraints

are forgotten and the *mandatory* nodes are ignored (since their Boolean value can trivially be assigned to True), the total number of configurations N_{total} is:

$$N_{total} = 2^{N_{nodes} - N_{mandatory}} = 2^{N_{features} + N_{contexts} + 2 - N_{mandatory}} = 2^{21 + 17 + 2 - 12} \approx 2.6e^8$$

Where N_{nodes} is the number of unified nodes, $N_{features}$ the number of features, $N_{contexts}$ the number of contexts, $N_{mandatory}$ the number of mandatory relationships. Even while overestimating the number of valid configurations, N_{total} is overwhelmingly higher, and grows even faster than the number of valid configurations in the number of unified nodes. Covering array generation that ignores constraints will nearly never find one single valid test case, and the regeneration approach is doomed to fail.

Instead, we use a CIT sampling technique that takes constraints into account directly in its operation. The algorithm was mainly inspired by the paper "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach", by Cohen et al. [21].

The research in the paper was made in the context of software product lines, but aims at a more general kind of system: "Highly-Configurable Systems". The algorithm they present generates a covering array with a one-row-at-a-time CIT greedy approach, upgraded by an innovating idea: directly taking into account the constraints of the systems during the discovery of the covering array. To achieve that, a SAT solver is used to prune the search space of the CIT problem while constructing a new test case (adding a new "row"), and the algorithm avoids creating any invalid test case in the final covering array with respect to the system's constraints.

It makes use of the ability of SAT solvers to determine whether a CNF Boolean expression is satisfiable or not, given a partial assignment of the literals within this CNF Boolean expression. For example, $(a \vee b) \wedge (a \vee c)$ is a satisfiable formula. However, the question of interest here is whether it is still satisfiable given $a = False$ and $c = True$.

In order to adopt a greedy CIT approach using a SAT solver, we first need to know how to use a SAT solver with a feature-based context-oriented system. This is the topic of the following section.

5.1.4 Adaptation to SAT solvers

The CIT algorithm is executed with the unified model we defined earlier as the input. Hence, we must adapt that unified model to SAT solvers. Fortunately, we already possess all the tools required.

The SAT solver needs to determine whether a partial test case is still valid or not. Coincidentally, in our case, the partial test case is already a partial configuration of unified nodes whose value is either True or False. In other words, this partial configuration is an assignment of truth values to literals.

The constraints of the unified model also directly relate to these literals, as they consist of feature diagrams' constraints (from the context and feature models) and equivalence dependencies (from the mapping between contexts and features). As seen in Section 2.1.3, we can convert feature diagrams' constraints to CNF Boolean expressions. Secondly, equivalence dependencies can be trivially converted to CNF Boolean expressions:

$$(A \Leftrightarrow B) \iff (-A \vee B) \wedge (A \vee -B)$$

The unified model's constraints convert to a set of CNF rules. Additionally, a partial test case is a partial assignment of truth values to literals. It is thus compatible with a SAT solver.

5.1.5 Greedy CIT-SAT algorithm in theory

We covered all the adaptations and conversions needed to a greedy CIT-SAT algorithm with a feature-based context-oriented system. This section analyzes the operation of the complete algorithm, adapted to our needs, through a pseudo code shown in Algorithm 1. As mentioned before, it was inspired by the work of Cohen et al. [21].

Algorithm 1: Greedy CIT-SAT algorithm

```
input : unifiedModel
output : coveringArray
1   createSATSolver(unifiedModel)
2   initialize(uncoveredPairs)
3   coveringArray =  $\emptyset$ 
4   numCandidates = 30
5   while uncoveredPairs  $\neq \emptyset$  do
6       testCasePool =  $\emptyset$ 
7       for count = 1 to numCandidates do
8           testCasecount = generateEmptyCase()
9           variant = selectBestVariant()
10          insertVariant(testCasecount, variant)
11          remainingFeatures = shuffleRemainingFeatures()
12          for  $f \in \textit{remainingFeatures}$  do
13              sat = false
14              tries = 1, maxTries = 10000
15              while !sat do
16                  variant = selectBestVariantForFeature(f)
17                  sat = checkSAT(testCasecount, variant)
18              end
19              insertVariant(testCasecount, variant)
20          end
21          saveTestCase(testCasecount, testCasePool)
22      end
23      saveBestCandidate(testCasePool, coveringArray)
24      update(uncoveredPairs)
25  end
```

Line 1 to 4 initialize the variables. *uncoveredPairs* is a list containing all pairs of variants to cover (line 2), and we will add new test cases to the covering array until all pairs are covered (line 5). However, all pairs are not just any possible combination of variants: we must take the constraints into account. For every possible pair, we add it to *uncoveredPairs* only if the constraints are still satisfiable given that pair, using the SAT solver (not shown).

Before adding any new test case to the covering array, we compute a total of *numCandidates* potential test cases (line 7) added to a *testCasePool* (line 21), and save only the best candidate among them in the final covering array (line 23).

The process of finding a new test case is defined between line 8 and 25. We first select the best feature and its variant which we add to the new test case (line 9 and 10). In our case, the best variant is the one that covers the most pairs in *uncoveredPairs*, or in other words, which is the most present in these pairs.

Ties are broken randomly, causing nondeterminism to appear in differing runs of the algorithm.

Once we added a first variant to the test case, we will add a variant for each remaining feature in a random order (line 11). Again, we select the best variant for a feature (line 16), and ties are broken randomly. This time, the "best" is determined by the number of uncovered pairs this variant would cover if we added it to the current test case. However, it can be inserted in the test case (line 19) only if adding it to the test case does not make the constraints unsatisfiable (line 17), and we will iterate over other variants if it does (line 15 to 18). The pruning of the CIT problem's search space is done here: we construct a test case which is valid at all times according to the constraints.

Concerning the variables, *numCandidates* was set to 30. Cohen et al. [21] advise fixing it to 50, but our experiments show that increasing it too much has no impact on the results, other than increasing substantially the computation time. We compromised at 30 to keep the quality of the solution while reducing the computation time. Our guess is that our study case is not big enough, and the value of 50 would yield better results with bigger models, while maintaining a relatively low computation time.

5.2 Summary

We solved several issues by defining an alternative modelling of a feature-based context-oriented system, the unified model. Thanks to it, we can use a greedy CIT approach combined to SAT solving in order to handle the highly-constrained *unified* model. We can generate covering arrays for feature-based context-oriented systems, which is how we reduce the number of possible test cases to be considered.

With this approach in mind, Chapter 6 presents our actual implementation, and the concrete testing methodology we propose.

Chapter 6

Implementation and results of the testing methodology

We found the core idea of a testing methodology, which is to generate covering arrays, and explained how to reconcile our system with the CIT sampling techniques used to generate them. We will now propose a testing methodology that builds around this idea. As our goal in this master thesis is to develop a testing methodology tailored to feature-based context-oriented systems, we will present the methodology in parallel to its implementation with the RubyCOP language framework. We will end this chapter with some results of this methodology when applied to our case study.

6.1 Testing methodology

The diagram in Figure 6.1 depicts the interactions between the different actors of our testing methodology. A solid arrow represents an interaction between two entities, like a data propagation or a method call. Feature-based context-oriented systems are built on top of the RubyCOP language framework, which is what the dotted arrow represents.

From the point of view of the user, using the proposed methodology is simple. While creating a feature-based context-oriented system, if his system is valid (consistent context/feature models and mapping), he can call the method *CITSAT()* from the *CITSAT algorithm* module. Then, he receives a polished covering array, which he can use to create efficient test suites and to find errors in his code.

To meet the developer's expectations, we will use the greedy CIT-SAT algorithm defined earlier. As a prerequisite to use it, we first need to extract the data from

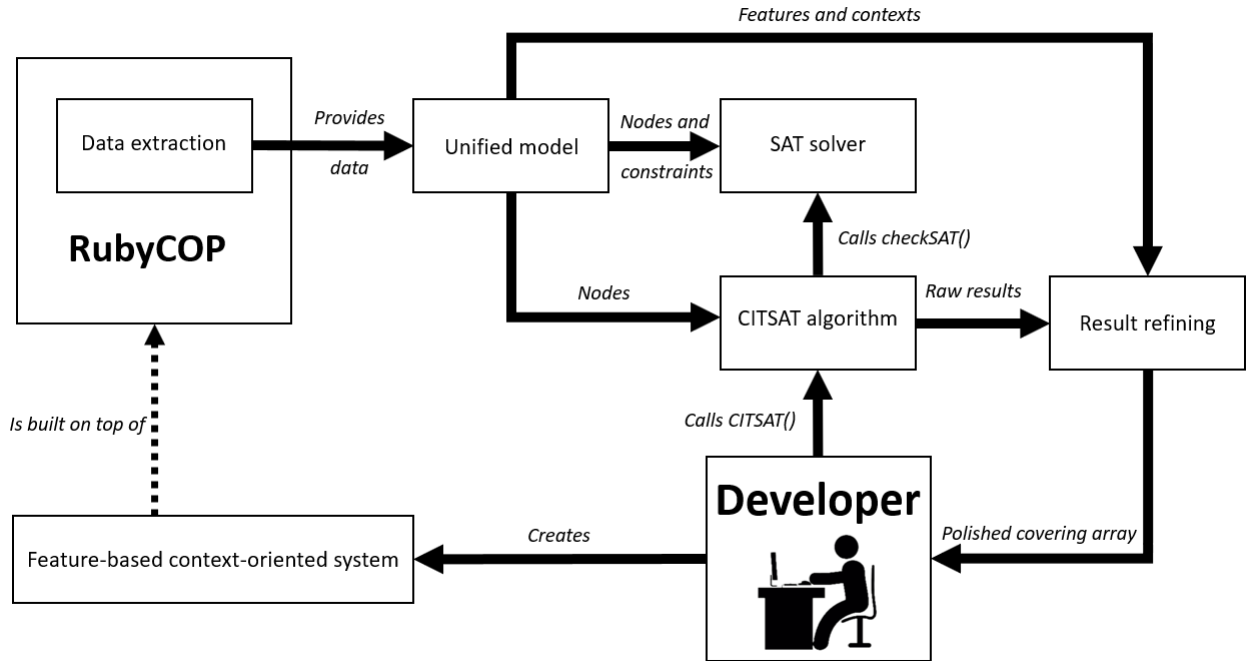


Figure 6.1: Diagram of the different modules and their interactions

the system, create a unified model and instantiate an appropriate SAT solver. After running the algorithm, we refine the results to clarify them and help the developer to use them. Each step is handled by a different module. In the following sections, we detail each of these steps: their related module, and technical details about how we implemented it.

6.1.1 Data extraction

The developer created a valid feature-based context-oriented system and wants a covering array for his system. The first step is to extract data from his system, in order to create a unified model.

Our objective, while extracting data, is to be as minimally invasive as possible. To this end, we limit the information we need to: the constraints in the feature model, the constraints in the context model, and the mapping. As all contexts originate from context model, all contexts are linked through relationships. Thanks to that, a list of the context model's constraints is enough to deduce the list of all possible contexts. The same is true for the features.

We extract them by writing formatted text files, which will be the bridge between RubyCOP and the rest of our implementation, coded in Python (version 3.7).

For clarity, we show again in Figure 6.2 the Entities part of the RubyCOP architecture, which was detailed in Section 2.2.2.

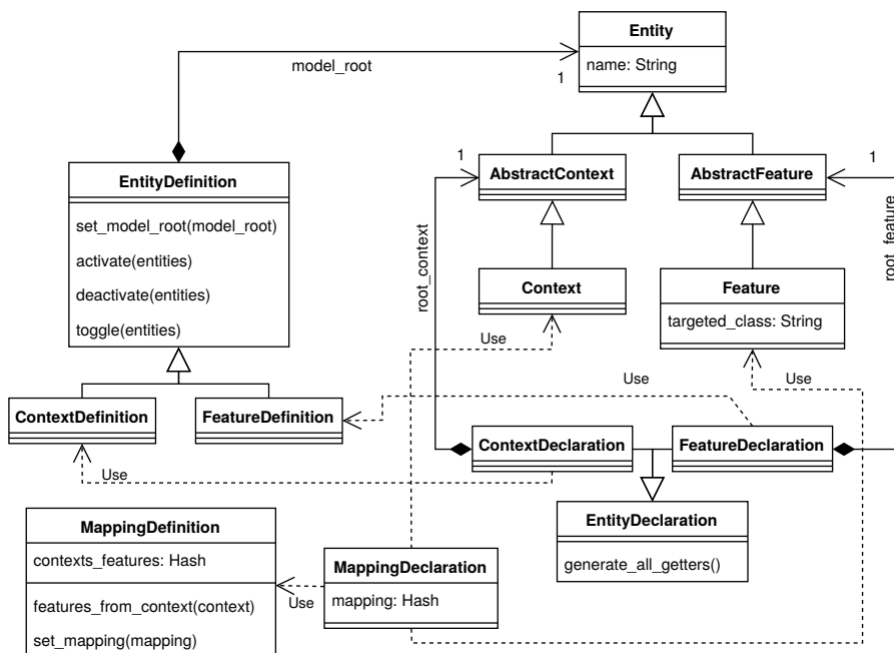


Figure 6.2: Zoom on the Entities part of the implementation architecture, extracted from "Implementation of a Feature-Based Context-Oriented Programming Language" [6]

To retrieve the models, we access the classes *AppFeatureDeclaration* and *AppContextDeclaration*, which extends the Entities *FeatureDeclaration* and *ContextDeclaration*. These Entities have an access to the roots of the feature and context model, and these roots are themselves instances of *Entity*. Every *Entity* object keeps track of its relations with other *Entity* objects, and the type of these relations. By iterating over these relations, we can find all the constraints of the context and feature models. We write them with a specific format in the files *contexts.txt* and *features.txt*, respectively.

To retrieve the mapping, we modified the Entity *MappingDeclaration*. We added the method *mapping_list*, whose simple goal to serve as a getter for the

mapping attribute, a hash data structure containing every dependency between contexts and features. It returns a formatted text containing the information contained in *mapping*, which we will write in the file *mapping.txt*.

The methods to write these three files and the modification of *MappingDeclaration* can be found in the Appendix, *Data extraction* module. The following modules are coded in python, and their files will be added to the code appendix of this thesis directly.

6.1.2 Data parsing

The role of the *Unified model* module is to parse the information from *contexts.txt*, *features.txt* and *mapping.txt*. It stores the complete information and processes it, notably in the form of a *unified* model.

This module's code can be found in the code appendix of this thesis, in the file *UnifiedModel.py*. The text files generated by applying our testing methodology to our case study can also be found, under the names *feature.txt*, *context.txt* and *mapping.txt*.

6.1.3 SAT solving

To generate a covering array, we need the CIT-SAT algorithm, and for this algorithm, we need a SAT solver. To this end, the *SATsolver* module extracts the list of *unified* nodes and the list of constraints from the *Unified model* module, instantiates a SAT solver and serves as the interface to it. Its main method is *checkSAT()*, which checks whether the constraints remain satisfiable considering a given partial configuration.

In order to instantiate a SAT solver, we use PySAT, a recent library (<https://pysathq.github.io/>) [37] which implements multiple state-of-the-art SAT solvers such as Minisat, Glucose, CaDiCaL, ...

Our choice fell on Glucose 3 (<https://www.labri.fr/perso/lsimon/glucose/>), a SAT solver based on MiniSAT (an open-source SAT solver) and specific discoveries [31][38]. That SAT solver performed greatly in numerous yearly contests organized by "The International SAT Competition" [39]. Since 2010 until January 2021, SAT solvers based on Glucose won more than 25 medals [39].

The code of this module can be found in the code appendix of this thesis. *SATSolver.py* contains the SAT solver interface, and *ConstraintsToCNF.py* contains the conversion of the various constraints to CNF rules.

6.1.4 Greedy CIT-SAT algorithm implementation

The *CIT-SAT algorithm* module's role is to implement the greedy CIT-SAT algorithm we defined in Chapter 5. The algorithm generates a covering array for a *unified* model, which is ultimately a covering array for its corresponding feature-based context-oriented system. The actual implementation closely reflects the pseudo code shown in Section 5.1.5, Algorithm 1, and its operation will not be detailed any further.

The algorithm in itself does not need to know what constraints the system is subject to, as they are handled by the *SAT solver* module. The only needed input is the set of *unified* nodes, from the *unified* model, which he retrieves from the *Unified model* module.

During its execution, it will repeatedly call *checkSAT()* to determine whether a selected variant makes the current test case invalid or not. After its execution, it will send the completed covering array to the *Result refining* module.

As a reminder, this algorithm is nondeterministic. To handle it, we use the library Random (documentation at <https://docs.python.org/3/library/random.html>).

The code of this module can be found in the code appendix of this thesis, in the *CITSAT.py* file.

6.1.5 Result refining

We obtained raw results: an unrefined covering array originating from the *CIT-SAT algorithm* module. The *Result refining* module's role is to help the developer by post-processing the covering array and presenting it in a more user-friendly way.

We created a covering array for a *unified* model. Naturally, the first step is to separate contexts and features, and that is why the module retrieves the list possible contexts and features from the *Unified model* module.

However, the covering array remains very hard to read. Let us take the example of our case study. Thanks to the preceding modules, we generated a covering array consisting of 10 test cases for our system. A test case is a set of (de)activated contexts and features, and we have a total of 18 contexts and 21 features (by counting the roots of their feature diagram, which is a part of the *unified* model). A single test case is already hard to read, because it consists of a list of 18 (de)activations. Creating 10 independent test cases, where we have to simulate 10 totally separate set of contexts (180 contexts (de)activations !), is even longer and arduous.

To clarify the covering array, we can extract all *commonalities* from the test cases and inform the developer that these *commonalities* should always be active. In our case, we remove 3 contexts and 8 features.

Then, we can list only the activated contexts and features, and not the deactivated ones. In our case, at this point, the number of activated contexts will range from 2 to 8, which is already much more readable. A part of such a covering array, which we call a semi-refined covering array, is shown in Table 6.1.

Test Case Number	Activated contexts
	Activated features
1	'Teen', 'Loud', 'In Pocket'
	'Match', 'ProfilePicture', 'Vibrator', 'GroupFeature'
2	'Connection', 'Sound', 'Good Connection', 'Day', 'Handle Photos', 'Out Pocket', 'Device Capabilities', 'Adult'
	'Standard', 'Alarm', 'Search', 'Description', 'Vocals', 'Mode', 'Photos'
3	'Connection', 'Sound', 'Day', 'Teen', 'Out Pocket', 'Device Capabilities', 'Bad Connection'
	'Match', 'Alarm', 'ProfilePicture', 'Lite', 'GroupFeature', 'Vocals', 'Mode'

Table 6.1: Three test cases of a semi-refined covering array

However, we can refine even further. In the last two test cases, we observe a very interesting phenomenon: the two of them have 5 activated contexts and 3 activated features in common. This is often the case between two test cases, and this

redundant information could be synthesized thanks to an alternative representation. Instead of showing the list of activated contexts/features, the displayed result will be the list of contexts/features to (de)activate compared to the previous test case.

In order to maximize this effect, we can order the test cases. To order test cases, we define the *distance* between two test cases to be equal to the number of activated (or deactivated) contexts in the first test case which are deactivated (or activated) in the second one.

Let us imagine an empty array, which will become a sorted covering array. The first test case added to this array is the one with the smallest number of activated contexts (other criteria can be used). Following the first, with the last added test case as the reference, each new test case is the *closest* among those not already sorted.

The result of such an ordering is the final way we will represent the obtained covering arrays. An example will be shown in the next section.

The code of this module can be found in the code appendix in this thesis, in the *ResultRefining.py* file.

6.2 Result

We described every part of the proposed testing methodology and the corresponding implementation. The final result is a 2-way covering array represented in a user-friendly manner. Applied to this master thesis' case study, the methodology produces a covering array of size 6 shown in Table 6.2. As we mentioned previously, the algorithm at the core of the proposed testing methodology is nondeterministic, and every run will produce a different covering array. However, the size of the generated covering array was consistently 6. The result includes a list of *commonalities* for the context model, which are:

Environment, Contexts, Age

It also includes a list of *commonalities* for the feature model, which are:

*Textual, AddSystem, Friend, MessageType, FriendName,
Features, NotificationSystem, ContactList*

Test Case Number	Contexts to activate	Contexts to deactivate
	Features to activate	Features to deactivate
1	'Adult', 'InPocket', 'Loud'	X
	'Vibrator', 'Search', 'Descr'	X
2	'Bad', 'Teen', 'Connection'	'Adult'
	'Lite', 'ProfilePicture', 'Mode', 'Match'	'Search', 'Descr'
3	'Good', 'DCapabilities', 'HandlePhotos'	'Bad'
	'Photos', 'Standard'	'Lite'
4	'Adult', 'Day', 'OutPocket', 'Sound'	'InPocket', 'Teen', 'Loud'
	'Vocals', 'Search', 'Descr', 'Alarm'	'Vibrator', 'ProfilePicture', 'Match'
5	'Bad'	'Good', 'Handle Photos'
	'Lite'	'Photos', 'Standard'
6	'Teen'	'Adult', 'Bad', 'Connection'
	'ProfilePicture', 'Match'	'Lite', 'Search', 'Mode', 'Descr'

Table 6.2: Refined covering array for our case study.
'Description' was shortened to 'Descr' and 'DeviceCapabilities' to 'DCapabilities' to shorten the test cases where they were mentioned.

Thanks to the user-friendly refining, we only have to (de)activate four contexts for four of the six test cases, the other two requiring to (de)activate three and seven contexts. With this result, creating a continuous test suite is easy and straightforward. The cost of creating a test suite is minimized.

Moreover, the developer can invent scenarios fitting the test cases easily. For example, we could imagine the first test case representing a working man commuting to work. The second test case would be his son, at home, and he suffers from a bad wi-fi. The third would be his rich friend, with good wi-fi and a better smartphone. Small variations in context are easy to imagine.

One could ask, with reason, why we chose to display the (de)activations of features, since they are commanded by contexts. Indeed, in the covering array shown in Figure 6.2, we can infer the (de)activations of features from those of contexts. There are two reasons: first, there could be an oversight in the mapping. The case where a feature is never activated through the mapping exists, and does

not necessarily cause an error (e.g. if this feature is optional). Even before coding a test suite, the developer can see a discrepancy in the result, as a feature will be activated or deactivated without any context commanding it.

The second reason is clarity. As a developer, we find important to see what configurations we actually test. The errors do not come from the contexts, as they are not related to any classes or code from the developer. They come from the behaviour and interactions induced by the (de)activation of features, which directly represent classes created by the developer. If the second test case passes the tests without any problem, but an error occurs while testing the third test case, then we have a high chance of finding it by examining the interactions involving the *Photos*, *Standard* and *Lite* features.

You can replicate the results by executing *main.py*, which is in the code appendix of this thesis, with Python 3.7 or higher and PySat installed (see <https://pysathq.github.io/installation.html>).

In summary, we implemented the greedy CIT-SAT algorithm defined in Chapter 5, and integrated it in a complete testing methodology. From a valid feature-based context-oriented system, we generate a refined covering array, suitable for a developer to read and use. The results on the proposed case study are promising. In the next chapter, we will analyze the strengths and threats to validity of the proposed testing methodology.

Chapter 7

Validity

We saw in Chapter 5 what led us to the proposed testing methodology, and why it is suitable to feature-based context-oriented systems. We proposed and implemented a comprehensive testing methodology in Chapter 6. We will now analyze the actual strengths and weaknesses of this methodology. We will see what it can offer in terms of practicality, rapidity, efficiency, and incrementality, notably when scaling up to bigger systems. We will analyze most of these by applying the proposed methodology to our case study.

Finally, we will discuss the threats to the validity of our solution, and some of its limitations.

7.1 Practicality

The proposed testing methodology is easy to use. The developer does not need to tinker with the RubyCOP language framework, and only needs an access to the *CIT-SAT algorithm* module to generate a covering array.

Moreover, we saw that the results could efficiently be used to create continuous, readable test suite. The refined covering array offers tractable test cases, and the amount of work to create a test suite (i.e. the number of (de)activations of contexts the developer must simulate) is minimized.

From a creative point of view, with the help of the refined covering array, the developer can more easily imagine scenarios for each test case. These stories can be used to discover more contexts and features to implement in his application, by adding details to it.

7.2 Rapidity

Generating results for our case study takes 6.99 seconds on average (tested on 100 runs). The maximum recorded computation time is 7.65 seconds and the minimum one is 6.55 seconds. Overall, the computation time is very stable, with a maximum variation of approximately 15%.

Adding more features or constraints naturally increases the computation time. However, the work of Cohen et al. [21] showed that the computation time of greedy CIT approaches associated to SAT solving does not scale exponentially when scaling up to bigger systems, with more features or constraints. In this master thesis, we proposed a testing methodology in a static context where time is not of the essence, and we did not model the variations in computation time when scaling up to bigger feature-based context-oriented systems. We will need a more thorough study to assess the scalability of the computation time if we want to integrate it in a dynamic testing approach.

7.3 Efficiency

As mentioned previously, we consistently find a covering array of size 6 for our case study. We simulated bigger systems with a simple approach: we added *optional* features to the *unified* model. By adding 15, which multiplies all possible configurations by 2^{15} , we consistently obtained a covering array of size 12. Adding 20 also resulted in a covering array of size 12. As expected, the covering array's size grows logarithmically in the number of features [20], which is a good point when scaling up to bigger systems.

The only downside to adding more contexts and features is that, as the covering array remains small, there are more contexts/features to (de)activate between the generated test cases. However, this is not really a problem, as we can duplicate test cases if needed: instead of activating both A and B in the n th test case, it only activates A, and we insert a n th bis test case which activates B. Augmenting artificially the covering array in order to keep a readable continuity between the test cases could be considered in the future.

7.4 Incrementality

This testing methodology can be integrated in an incremental testing approach. If you already have a test suite for a feature-based context-oriented system, and you

want it to reuse it when you expand your system with new contexts and features, it is possible. Instead of generating a covering array from nothing in the CIT-SAT algorithm, we can initialize it with the previously generated test cases augmented to fit the new contexts and features in the best way possible (the best way possible being the one that covers the most uncovered pair of variants, as usual in a greedy approach).

Even if it is done at the cost of efficiency, as the final covering array will be larger than a new generated one, it allows the reuse of an old test suite.

7.5 Threats to validity

We identified three threats to the validity of our solution. The first one is the generalization to other case studies. We cannot be sure that our testing methodology was not specifically fit to our case study. It could have difficulties adapting to totally unrelated subjects.

The second threat is observation bias. We found results and interpreted them in a way that seemed logical to us. The practicality of our solution was assessed according to our testing standards, but we can be biased and it doesn't necessarily meet the expectations of every developer.

The last one is the most general. We found a solution and developed a testing methodology around it, by adapting techniques from another field of study, CIT. However, feature-based context-oriented programming is an emerging paradigm under active development. We cannot be sure there are no other studies, unrelated to combinatorial interaction testing, that can also be adapted to it and that offer a better solution.

7.6 Limitations

The usefulness in using covering array to test highly-configurable systems was proved by many studies to efficiently find certain types of faults [21] [20], or to provide good code coverage [21] [20] [36]. However, it cannot cover all errors. After building a complete test suite from a covering array, errors can remain unnoticed in the system.

A second limitation to our testing methodology is naturally its inability to analyze and evaluate a specific feature or test case. We considered the features of a feature-based context-oriented system as a whole, and the proposed testing methodology assures the cover of all the interactions between them, but goes no further.

Finally, from a technical point of view, there could be problems in maintaining our solution while developing feature-based context-oriented programming. Our implementation relies on the RubyCOP language framework, and more specifically in accessing specific Entities within its architecture. If the architecture of RubyCOP were to change drastically, we would have to adapt our solution to the new architecture.

Chapter 8

Future work

The contributions were all reviewed, the solution was verified and its actual use was analyzed. Now, what is left to do is look to the future and see what remains to be done.

We identified four different topics of future work. The first three are about improving the current solution. In particular, we will see that we can better integrate, optimize and maintain our solution. The last part is about tackling the remaining unsolved problems, and is the most interesting research topic.

8.1 Better interface

As mentioned previously, SAT solving was explored in a preceding master thesis [5]. There are certainly other functionalities involving the use of a SAT solver to be discovered in feature-based context-oriented programming, and to support them, we could build a comprehensive SAT solver interface which allows to exploit more of the SAT solver possibilities.

In the same spirit, we could create a complete library in RubyCOP that spans all the possibilities analyzed in Chapter 7. This library should connect the python code to RubyCOP, for example by making use of PyCall (<https://github.com/mrkn/pycall.rb>), pipes (http://www.decalage.info/python/ruby_bridge), or other interoperability mechanisms.

However, as very little interaction is needed between the two languages, we could just automatically passively create the files needed for the algorithm (see section 6.1.1) while coding in RubyCOP. With one click, running the Python code and generating the test cases the way we need them is largely efficient enough.

8.2 Optimization

Two variations of the greedy CIT-SAT algorithm are presented by Cohen et al. [21]. One optimizes the number of generated test cases and the other the speed of execution [21], both by exploiting more efficiently the SAT solver. Optimizing the number of test cases has a limited impact in our case, as the number of features and contexts remain at a human-size (less than hundreds of features and contexts).

On the other hand, optimizing the speed of execution could be useful. Their result shows that this optimization can divide by up to three the computation time, for some test cases. Currently, the algorithm takes approximately 7 seconds to run, as seen in Chapter 7. A faster execution would allow a permanent and passive update of the generated test cases without perturbing a developer that is adding features to its model. It would dynamically help him in this task without computing overload.

These two optimizations, especially the second one, would be beneficial when scaling up to build larger systems with more than just a few contexts and features.

In Chapter 6, we defined a specific ordering in our covering array. Other studies in Software Product Lines have already explored this idea, and they defined a prioritization used to sort/generate covering arrays, optimizing coverage criteria or weights assigned to features [22]. More ideas could be extracted from the field of SPL testing to improve our generation of test cases, and our ordering.

8.3 Maintenance

Currently ongoing research on feature-based context-oriented programming tries to explore more complex mappings, with more complex interactions between contexts and features than a mere one-to-many mapping. For now, we can easily convert all of the constraints found in the mapping into CNF clauses. If the mapping were to change, we would have to convert these new constraints into CNF clauses as well.

More generally, if the models or the architecture used in RubyCOP were to change drastically, the data extraction might be at risk. Indeed, we currently rely on methods implemented deep inside this architecture to retrieve information from it, and it is one of the limitations we discussed in the previous chapter. In the future, building a formal interface in the RubyCOP framework which allows a visibility of the system's state (context and feature models, mapping, etc), no

matter its architecture, would be more robust towards future evolution of the approach.

8.4 Complete the testing methodology

As we saw in Chapter 4, there are two main issues to develop a testing methodology. The first one, which we solved here, was to address the exponential number of possible test cases to be considered. The second stems from the architecture and the mechanisms of the framework itself, such as the *proceed* mechanism, which allows to dynamically refine existing features with new behaviour. Considering that architecture, we still do not know how to test a specific feature or test case efficiently. Obviously, we must address this issue in order to complete our testing methodology (see Section 4.3, where we elaborated more on this issue).

Our implementation can naturally be improved further, but works as is. The next step is thus to advance the testing methodology, and solving the remaining issues is the most interesting progress from the point of view of research.

8.5 Summary

We saw four different ways to advance and build the testing methodology. The next step of the research should be focused on the fourth issue, which is searching for a solution reconciling our current testing methodology and a dedicated testing approach for the framework's powerful language mechanisms, in particular the *proceed* mechanism. Once all issues are solved, we can come back and polish the results we obtained in this master thesis in order to obtain a complete and finished tool.

Chapter 9

Conclusion

Feature-based context-oriented programming is a relatively new paradigm in the field of context-oriented programming, and is constantly evolving. A part of this evolution is developing tools for developers creating feature-based context-oriented systems built on top of the RubyCOP language framework.

These systems can include critical applications, and testing them before release is mandatory.

With regard to this concern, our main contribution in this master thesis is to address the issue of the exponential number of possible test cases to be considered in feature-based context-oriented systems and to propose an appropriate testing methodology implemented in parallel to RubyCOP.

We succeeded in providing a scalable, tractable set of test cases to developers. The cost of creating a test suite was minimized. This test suite, originating from a 2-way covering array, is assured to cover all possible interactions in the system and to find faults efficiently.

Finally, we saw that the development in the testing field is only starting. While we solved one issue, we still have no way of testing a specific feature or test case efficiently. Indeed, there are issues coming from the architecture and the mechanisms of the RubyCOP framework itself, such as the *proceed* mechanism. These issues have to be addressed before we can build a coherent testing methodology, with this master thesis' contribution as the first milestone.

Bibliography

- [1] Benoît Duhoux. L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte. Master's thesis, Université Catholique de Louvain-la-Neuve, 2015-2016.
- [2] David Sarkozi. Programming context-aware features in ruby. Master's thesis, Université Catholique de Louvain-la-Neuve, 2015-2016.
- [3] Alexandre Kühn. Reconciling context-oriented programming and feature modeling. Master's thesis, Université Catholique de Louvain-la-Neuve, 2017.
- [4] Hoo Sing Leung. Visualisation of contexts and features in context-oriented programming. Master's thesis, Université Catholique de Louvain-la-Neuve, 2018-2019.
- [5] Alexis Van den Bogaert. Consistency management of contexts and features in context-oriented programming language with sat solving. Master's thesis, Université Catholique de Louvain-la-Neuve, 2020.
- [6] Benoît Duhoux, Kim Mens, and Bruno Dumas. Implementation of a feature-based context-oriented programming language. *COP'19*, 2019.
- [7] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, July 2009.
- [8] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie Mellon University, 1990.
- [9] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. *Proceedings of the International Conference on Algebraic Methodology and Software Technology*, 2008.
- [10] M. Mendonça, A. Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. *Software Product Lines, 13th International Conference*, August 2009.

- [11] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. Modeling and managing context-aware systems' variability. *IEEE Software*, 34, 2017.
- [12] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. *Proceedings of 12th International Software Product Line Conference*, October 2008.
- [13] Aitor Murguzur, Rafael Capilla, Salvador Trujillo, Óscar Ortiz, and Roberto E. Lopez-Herrejon. Context variability modeling for runtime configuration of service-based dynamic software product lines. *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools*, 2:2–9, 2014.
- [14] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc and Ubiquitous Computing*, 2007.
- [15] H. Chen. *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD thesis, University of Maryland, Baltimore County, 2004.
- [16] T. Winograd. Architectures for context. *Human-ComputerInteraction (HCI) Journal*, 16, 2001.
- [17] Unai Alegre, Juan Carlos Augusto, and Tony Clark. Engineering context-aware systems and applications : a survey. *Journal of Systems and Software*, 2016.
- [18] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. *Proceedings of the 8th International Workshop on Context-Oriented Programming*, 2016.
- [19] Vahid Garousi and Mika Mäntylä. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, April 2016.
- [20] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 23, July 1997.
- [21] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 34(5), September/October 2008.

- [22] Xavier Devroey, Maxime Cordy, Gilles Perrouin, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards statistical prioritization for software product lines testing. *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, 2014.
- [23] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. *Proceedings of the 17th International Software Product Line Conference*, 2013.
- [24] Robert M. Hierons, Miqing Li, Xiahui Liu, Jose Antonio Parejo, Sergio Segura, and Xin Yao. Many-objective test suite generation for software product lines. *ACM Transactions on Software Engineering and Methodology*, 2020.
- [25] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.
- [26] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge. Constructing test suites for interaction testing. *Proceedings of the 25th International Conference on Software Engineering*, May 2003.
- [27] K. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Math.*, 2004.
- [28] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *J Res Natl Inst Stand Technol*, 2008. 113(5): 287–297.
- [29] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.
- [30] Niklas Een, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2007.
- [31] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*. Citeseer, 2009.
- [32] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. *International Conference on Principles and Practice of Constraint Programming*, 2007.
- [33] Benoît Duhoux, Kim Mens, and Bruno Dumas. Feature visualiser: an inspection tool for context-oriented programmers. *COP’18*, July 2018.

- [34] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. Dynamic visualisation of features and contexts for context-oriented programmers. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2019.
- [35] Benoît Duhoux and Kim Mens. A context and feature visualisation tool for a feature-based context-oriented programming language. *SATToSE*, 2019.
- [36] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iamino. Applying design of experiments to software testing. *Proceedings of the 19th international conference on Software engineering*, 1997.
- [37] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.
- [38] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.
- [39] The international sat competition web page. <http://www.satcompetition.org/>. Accessed: 2021-01-07.

Appendix

Data extraction module

```
def write_all_files()
  write_mapping_file()
  write_constraints_file(
    AppFeatureDeclaration.instance.root_feature(), 'features.txt')
  write_constraints_file(
    AppContextDeclaration.instance.root_context(), 'contexts.txt')
  puts "All files were written without any problem."
end

def write_constraints_file(root, file_name)
  relations = ""
  queue = [root]
  visited = []
  until queue.empty?
    node = queue.shift()
    next if visited.include?(node.name)
    visited.push(node.name)
    node.relations.each do |type, relation|
      additional_relation = "#{node.name}/#{type.to_s}/"
      relation.nodes.each do |child|
        queue << child
        additional_relation += "#{child.name}-"
      end
      relations += additional_relation[0..-2] + "\n"
    end
  end
  File.write(file_name, relations)
end

def write_mapping_file()
  File.write('mapping.txt',
```

```
        AppMappingDeclaration.instance.mapping_list)
end
```

MappingDeclaration code

```
class MappingDeclaration
  (...)
  def mapping_list()
    list = ""
    mapping.each {|contexts, features|
      contexts.each {|context| list = list + context.name + "-" }
      list = list + 'IMPLIES'
      features.each {|feature| list = list + "-" + feature.name}
      list = list + "\n"}
    return list
  end
end
```


UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

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