The Journal of Systems & Software 197 (2023) 111570

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Test scenario generation for feature-based context-oriented software systems ☆

Pierre Martou *, Kim Mens, Benoît Duhoux, Axel Legay

ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

ARTICLE INFO

Article history:
Received 21 February 2022
Received in revised form 18 October 2022
Accepted 23 November 2022
Available online 5 December 2022

CCS Concepts:
Software and its engineering: Software testing and debugging
Context specific languages
Software product lines

Keywords:
Context-oriented programming
Feature modelling
Dynamic software product lines
Software testing
Combinatorial interaction testing
Satisfiability checking (SAT)

ABSTRACT

Feature-based context-oriented programming reconciles ideas from context-oriented programming, feature modelling and dynamic software product lines. It offers a programming language, architecture, tools and methodology to develop software systems consisting of contexts and features that can become active at run-time to offer the most appropriate behaviour depending on the actual context of use. Due to their high run-time adaptivity, dedicated tool support to test such systems is needed. Building upon a pairwise combinatorial interaction testing approach from the domain of software product lines, we implement an algorithm to generate automatically a small set of relevant test scenarios, ordered to minimise the number of context activations between tests. We also explore how the generated scenarios can be enhanced incrementally when the software evolves, and how useful the proposed testing approach is in practice.

1. Introduction

Context-Oriented Programming (COP) languages help developers build context-aware applications that, based on contextual information sensed from the surrounding environment, can select more appropriate behaviour at runtime. A wide range of COP languages have been proposed to ease the design and implementation of such applications (Costanza and Hirschfeld, 2005; Hirschfeld et al., 2008a; González et al., 2011; Salvaneschi et al., 2011).

Feature-Based Context-Oriented Programming (FBCOP) (Duhoux et al., 2019a,b) is a particular class of such languages that takes inspiration from Context-Oriented Programming (Hirschfeld et al., 2008b; Salvaneschi et al., 2012; Cardozo et al., 2011, 2014), Feature Modelling (FM) (Kang et al., 1990) and (Dynamic) Software Product Lines (DSPL) (Hartmann and Trew, 2008; Capilla et al., 2014b; Mens et al., 2017) and combines them into a single software development approach (Duhoux, 2022).

FBCOP systems are akin to software product lines as they can be seen as an implementation of a family of systems of which the current behaviour depends on the actual context of use. The possible behaviours of the system are described in terms of a feature model and the possible contexts in terms of a context model, using the same feature diagram notation. Which features are active at runtime depends on which contexts are currently active. A valid configuration of the context model describes a particular context of use, and triggers the selection of features specific to those contexts through a mapping model of contexts to features.

Whereas FBCOP languages provide better support for dynamic adaptation to context than traditional programming languages, traditional testing techniques meet their limits when applied to context-oriented applications. Dedicated techniques to generate test scenarios are needed to cope with the high variability of the environment (context) and how the application can change its behaviour (features) to situations in that environment. Such a test scenario defines a specific situation (set of active contexts) in which the application exhibits a particular behaviour (set of active features).

In addition to providing an overall architecture (Mens et al., 2016) and programming language (Duhoux et al., 2019a), we are investigating an encompassing design methodology and supporting development tools (Duhoux et al., 2018, 2019b) to help

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: pierre.martou@uclouvain.be (P. Martou), kim.mens@uclouvain.be (K. Mens), benoit.duhoux@uclouvain.be (B. Duhoux), axel.legay@uclouvain.be (A. Legay).

https://doi.org/10.1016/j.jss.2022.111570
0164-1212/

developers in building such software systems. An important part of this methodology, and focus of the current paper, is how to test feature-based context-oriented software systems. The five research questions (RQ) below capture the main contributions of this paper:

RQ1 *How to generate a pertinent yet tractable set of test scenarios for a given FBCOP application?* This RQ explores how to generate automatically a relevant set of contexts and their corresponding features, corresponding to a realistic situation worthwhile being tested.

To address RQ1, inspired by related work in the area of SPLs, we explore the use of pairwise *Combinatorial Interaction Testing* (CIT) to generate a small yet representative set of test scenarios that covers all valid combinations of pairs of contexts or features. We thus reduce the problem of generating test scenarios for FBCOP programs to one of computing the *covering array* of a highly reconfigurable system in the presence of constraints, granting direct access to an efficient greedy SAT-solving algorithm (Cohen et al., 2008) with logarithmic growth according to the number of features.

RQ2 *How to minimise the effort of simulating these generated test scenarios?* This RQ evaluates the effort required for a tester to set-up actual tests that simulate a generated set of test scenarios.

While the algorithm explored in RQ1 tries to minimise the number of tests needed to cover all interactions between pairs of contexts or features, the ordering of the different scenarios in the generated test suite may not be optimal. A software tester may prefer the scenarios to be ordered in such a way that only a minimal amount of reconfiguration of the context-aware program to be tested is required between each test scenario. Such a reconfiguration requires adapting or simulating the program via the activation and deactivation of certain contexts. Since FBCOP programs are modelled in terms of separate context and feature models, we can focus on the context model alone to define a *reconfiguration cost* which is the number of context activation switches needed to go from one test configuration to another. Our answer to RQ2 is then a greedy algorithm to reorder test scenarios in the test suite so as to minimise this cost, effectively reducing the effort for software testers to implement these tests.

Our next research question explores the effectiveness of our testing approach in practice, as well as the kind of errors it can help to discover, and is thus split in two parts:

RQ3a *How effective is our approach to find errors?* This RQ will evaluate if our approach is effectively able to find useful errors worth correcting.

RQ3b *What type of errors can we find using this testing approach?* Related to the previous question we are interested in finding out what types of errors our approach is best at discovering, e.g. design errors or programming bugs.

Whereas RQ3 focuses on the quality and types of errors that can be found, RQ4 assesses the usability of our testing approach to software developers who are not proficient in software testing:

RQ4 *How relevant and easy-to-use is the testing approach for developers?*

To address research questions RQ3a and RQ3b, we first illustrate our testing approach on a case study to highlight typical errors. We then complete this qualitative study and at the same time answer RQ4 through a broader experiment: we asked several developers to design, develop and test a small feature-based context-oriented application using our approach. Through a questionnaire we evaluated how effective and usable (strengths, weaknesses) they perceived our testing approach to be and what types of errors they could find with it. It turned out to be more useful to find conceptual errors in the design of their context and feature models than to find bugs in the code of their programs.

RQ5 *How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?* This RQ explores if a previous test suite can be reused to include new contexts and features when the system is modified.

Regarding RQ5, we want to avoid having to regenerate completely the test scenarios when new contexts or features are introduced as a program evolves to a new version. To avoid this, our approach consists of updating test scenarios that were already generated for the previous version of the evolved program. Assuming that this test suite already contains many of the new pairs that should be covered, Cohen's algorithm (Cohen et al., 2008) is fed with this candidate test suite to produce a small number of test scenarios covering the new pairs that are not yet covered. We propose two different update strategies and show that we can drastically reduce the cost of obtaining an up-to-date test suite by almost a factor 4 on our case study.

We implemented and evaluated each of these contributions on a small exploratory case study. Our initial experiments confirm the feasibility, efficiency and relevance of our testing approach, effectively addressing each of our five research questions. The five contributions combined provide a lightweight methodology and set of algorithms for creating effective test suites easily and efficiently while developing feature-based context-oriented systems.

Expanding upon an earlier paper (Martou et al., 2021), we largely rewrote and extended it on several aspects, most notably its validation with two additional research questions (RQ3 and RQ4). To answer these, we illustrate the use of our testing approach on a concrete case study, and also conducted an experiment where developers used our approach (new Section 7). Another goal of this extended paper is to elaborate upon the foundations of the feature-based context-oriented programming approach and methodology in much more detail, as well as its relation with other approaches such as feature modelling, dynamic software product lines and context-oriented programming (new Sections 3.1 and 3.2). In addition to that we added a new Subsection 3.4 describing our methodology to design such systems. This new methodology and the new exhaustive description of feature-based context-oriented programming clarify the need and place of our testing approach, and will serve as an important reference for future research on the topic. We also significantly reinforced our answer to RQ2 by adding a new contribution analysing clearly the advantages of our testing approach for the case of feature-based context-oriented systems (new Section 6.2), and further justified our choices by comparing our prioritisation criterion to other approaches in SPL testing (new Section 6.3). In the same spirit of validation, the current paper also discusses in greater depth the strengths and weaknesses of our approach, and possible future work.

The remainder of this paper is structured as follows. Section 2 describes the exploratory case study used as running example and validation. Section 3 introduces and illustrates our FBCOP approach on top of which we built our test scenario generation approach. Section 4 revisits each of our research questions and establishes our goals. Section 5 through 8 then explain and validate each of our five contributions in detail. Section 9 concludes the paper and presents interesting avenues of future work.

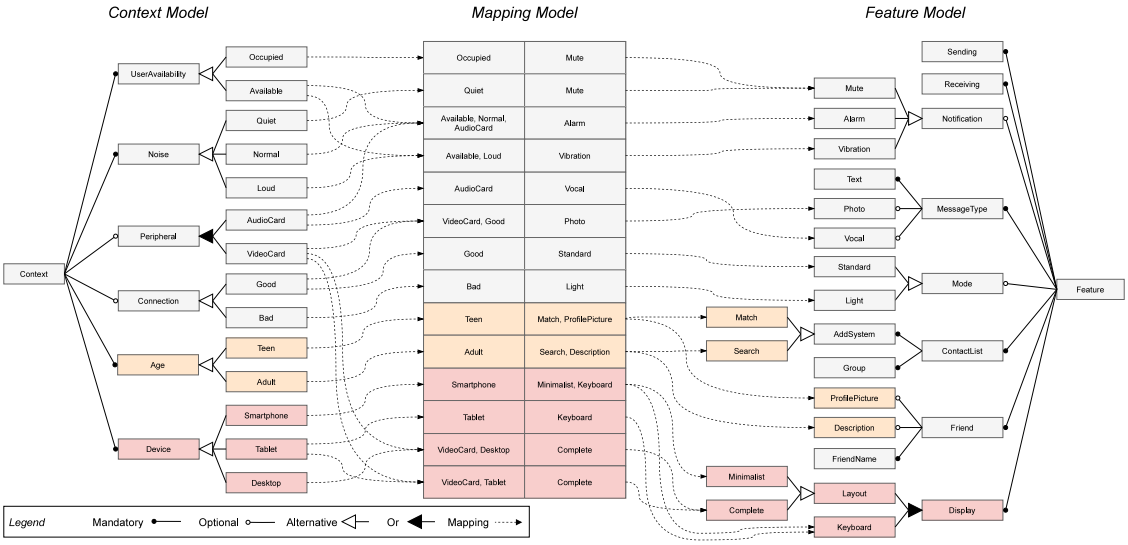

Fig. 1. Design of a prototype implementation of our smart messaging system. The context model is on the left, the feature model on the right and the mapping model in between them. The colours will be used in Section 8.2 to illustrate our test suite augmentation algorithm. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

2. Running example: a smart messaging system

Before explaining our FBCOP approach in more detail, this section introduces the case study that will be used as running example throughout the paper. More specifically, we implemented a prototype of a *smart messaging system*. This system allows users to exchange messages and is 'smart' in the sense that it can adapt or refine its behaviour to some contexts. The system's core functionality consists of *Sending* or *Receiving* messages to and from other users. By default, only *Text* messages are supported, though messages including other media types such as *Vocal* or *Photo* can be exchanged if the internet *Connection* is *Good* and the device running the program has the necessary *Peripheral* cards installed (*AudioCard* or *VideoCard*) that can support such media types.

Messages can also be exchanged between a *Group* of friends. Each *Friend* is uniquely identified by its *FriendName*. Depending on their *Age* (*Teen* or *Adult*), their profile can also be completed with either a textual *Description* or *ProfilePicture*, respectively. How new friends can be added to a conversation (either through a profile *Match* or *Search*) also depends on the users' *Age*.

Whenever users receive a new message from a friend, a *Notification* may be given via a sound *Alarm* or their device's *Vibration* mode. What notification mode is used depends on the device's *Peripherals*, the *User Availability* and the ambient *Noise* level. If the user is *Occupied*, notifications will be *Muted* altogether. The *Device* type (*Smartphone*, *Tablet* or *Desktop*) also affects other features of the smart messaging system, such as adapting the *Layout* of the information being *Displayed* or adding a virtual *Keyboard*.

The feature, context, and mapping models of this system, depicting the system's features and the contexts that trigger them, are shown on Fig. 1. Section 3 explains the underlying concepts of the design and implementation of such a system in more detail, and proposes a methodology to develop such systems.

3. Feature-based context-oriented software development

This section introduces the underlying principles and concepts of our feature-based context-oriented programming approach 3.2, its architecture 3.3 and supporting development methodology 3.4. Before doing so, we discuss a variety of related approaches, many of which motivated this new approach 3.1.

3.1. Motivation

A software product line describes a family of software systems that share a common set of features satisfying the specific needs of a particular domain. Products (software systems) of that family are produced by selecting and composing a set of features (configuration) corresponding to the needs of a particular client or variant of the system. Often feature modelling (Kang et al., 1990) is used to describe the possible features, together with their relations and constraints, of such a family. The features of a family describe the commonalities (features always present) and variabilities (features that may be present) of the different systems belonging to the family.

Dynamic software product lines (DSPL) try to address the need of dynamic software adaptation to provide the flexibility required by more dynamic environments and users (Capilla et al., 2014a). Their goal is that the software adaptations and product variations can be selected even at runtime (Bencomo et al., 2010). To model these systems feature modelling is often used and extended with additional information such as attributes and/or cardinalities (Baresi and Quinton, 2015).

For self-adaptive systems to become truly context-oriented, an alternative to extending feature models with additional attributes that may contain contextual information, is to model *explicitly* the contexts, based on which the features need to be adapted dynamically. Some research on software product lines has indeed proposed to use feature modelling not only to model the features of the software systems in a software product line, but also to model the possible contexts, together with their relations and constraints, on which (the selection of) these features depend (Hartmann and Trew, 2008; Acher et al., 2009; Fernandes et al., 2011; Capilla et al., 2014b, 2015; Mens et al., 2017).

Without being exhaustive, at the implementation level, many different mechanisms have been explored to dynamically reconfigure systems at runtime: Aspect-Oriented Programming (AOP) (Dinkelaker et al., 2010), Game Theory based modelling (Cordy et al., 2013a,b), Model-Driven Engineering (MDE) (Bencomo et al., 2008; Baresi and Quinton, 2015), combinations of AOP and MDE (Morin et al., 2009), Feature-Based Adaptation Mechanisms (Rosenmüller et al., 2011), specific languages (Ter Beek et al., 2020; ter Beek and Legay, 2019), Feature-Oriented Programming and Software Development (Kästner and Apel, 2011), and context-aware reconfiguration engines (Mauro et al., 2018).

Whereas some of the DSPL approaches mentioned above have started to separate explicitly the notions of contexts and features at the modelling level, this is not yet the case for most implementation mechanisms. Context-oriented programming (COP) (Hirschfeld et al., 2008b), however, *does* provide explicit programming level abstractions for representing contexts and features as first-class citizens in the language, together with dedicated language constructs to adapt dynamically the behaviour of context-oriented applications. Nevertheless, COP languages typically provide no support for *modelling* contexts and features. For that reason, Costanza started to explore the idea of using feature diagrams to model the features of COP programs (Costanza and D´Hondt, 2008).

Feature-Based Context-Oriented software development (Duhoux, 2022) takes these ideas yet a step further by combining the ideas of explicitly separating and modelling contexts and feature modelling from both the DSPL (Hartmann and Trew, 2008; Acher et al., 2009; Capilla et al., 2014b, 2015; Mens et al., 2017) and COP (Cardozo et al., 2014) worlds and integrating them into a single approach. In this approach we model *and* implement contexts and features as two separate feature models, together with a mapping describing what contexts trigger what features. The programming language also comes with an incremental modification mechanism allowing to implement features that behaviourally refine previously selected features. Features get selected automatically when the contexts that trigger them become active, and get activated and deployed if the set of selected features satisfies the constraints imposed by the feature model (if not, the configurations of the context and feature model get rolled back to a previously valid state).

In the following subsections we will elaborate on the principles, concepts, architecture and supporting methodology of this novel programming approach.

3.2. Principles and concepts

The three key notions of FBCOP are features, contexts and the context–feature mapping. In this subsection we describe each of them and how they are modelled.

3.2.1. Feature

Whereas Kang et al. define a feature as *"any prominent or distinctive user-visible aspect, quality, or characteristic of a software system"* (Kang et al., 1990), we will adopt a slightly more specific definition of features as *"implementation components, or parts thereof, that are visible and distinguishable to the end-user and which may be more or less relevant depending on the particular user or usage context"*. Such implementation features may address functional, user interface and/or data concerns. For example, the feature *Sending* in our messaging system addresses the functional concern of sending messages through the network as well as the user interface concern to add a "send" button and its interaction in the layout.

Features can be more fine- or coarse-grained, depending on their number of interactions with other components. For example, the different variants of the *Notification* mechanism are quite fine-grained as they concern only how users are notified of new messages. Features like the *Standard* or *Light* mode are examples of more coarse-grained features that have a broader impact on the entire system. Whereas *Standard* mode corresponds to a 'normal' usage of the system, *Light* mode implements a simplified version of the system where less information is displayed for each active feature.

Regarding their binding time (Kang et al., 1990), in our FBCOP approach, whereas the features (and contexts) are modelled at design time, and then implemented, they get (un)selected, (de)activated and (un)deployed at runtime since the features are triggered dynamically when new situations are sensed in the surrounding environment.

Following Kang et al. (1990), we model the features in terms of a feature model that captures the commonalities and variabilities of a system. As illustrated on the right-hand side of Fig. 1, a feature model can be visually represented with a feature diagram where the nodes represent the features of the system and the edges are the hierarchical and cross-tree constraints between these features. Hierarchical constraints can be *mandatory* (child features must always be present if the parent feature is selected), *optional* (child features may be present when the parent feature is selected), *or* (at least one child is required when the parent feature is selected) or *alternative* (exactly one child feature is required when the parent feature is selected). In addition to these hierarchical constraints, cross-tree constraints may exist: *exclusion* to ensure that certain features will never get selected simultaneously, or *requirement* to ensure that a target feature is already present in the system before the source feature can get selected.

In a feature model, all these constraints are needed to determine whether a particular configuration (selection of features) is valid. A valid configuration is a set of features that together satisfy the constraints imposed by the model. A valid configuration of the feature model on the right-hand side of Fig. 1 would be the set of features: *Feature*, *Sending*, *Receiving*, *MessageType*, *Text*, *ContactList*, *AddSystem*, *Search*, *Group*, *Friend*, *Description*, *Display*, *Layout* and *Complete*. If the *Search* feature were not selected, the configuration would no longer be valid since selecting the *AddSystem* feature implies that exactly one sub-feature must be selected.

3.2.2. Context

The relevance and need for certain features may depend on the particular user or context of use. This means that the behaviour of a system may vary depending on the information sensed from the surrounding environment in which it operates. This information can take the form of user preferences (a user's age, (dis)abilities), information from external services (weather conditions), or internal data about the device on which the system runs (remaining battery level or other sensor information) (Abowd et al., 1999; Thevenin and Coutaz, 1999; Coutaz et al., 2005). We thus define contexts as *"reifying any information sensed from the surrounding environment, through user interaction, or information received from sensors describing the external conditions and internal state of the device on which the system runs, to represent particular situations that trigger relevant features to adapt the behaviour of the software system"*.

Like features, contexts can also be modelled with a feature diagram (Desmet et al., 2007; Hartmann and Trew, 2008; Acher et al., 2009; Capilla et al., 2014b, 2015; Mens et al., 2017), but where the nodes now represent the contexts to which the system can adapt and the edges are the hierarchical or cross-tree constraints between these contexts. The left-hand side of Fig. 1 depicts a context model for our messaging system.

As for a feature model, a valid configuration of a context model is a selection of contexts that satisfies all the constraints imposed by the context model. An example of such a valid set of contexts is: *Context*, *UserAvailability*, *Available*, *Noise*, *Normal*, *Age*, *Adult*, *Device* and *Smartphone*.

It is worth noting that in our FBCOP approach, contexts (and features) are binary, meaning that they are either active or not. This is the case even for contexts based on continuous measures such as the *Noise* level. To distinguish between different noise levels, they are discretised into a partition of possible ranges, represented by alternative child contexts such as *Quiet*, *Normal* and *Loud*.

Table 1
Excerpt of the mapping model of our running example.

Contexts	Features
Occupied	Mute
Quiet	Mute
Normal, Available, AudioCard	Alarm
Loud, Available	Vibration

3.2.3. Context–feature mapping

Having defined the context and feature models, we still need to explain how to relate them. That is the purpose of the context–feature mapping. Whereas in this section we will detail and discuss the nature of the context–feature mapping in general terms, Section 3.2.4 will show its formal translation to propositional logic. We define the context–feature mapping as: *"a set of relations from the context model to the feature model to express what contexts trigger what features in order to adapt dynamically the behaviour of the running system"*. The table in the middle of Fig. 1 shows the context–feature mapping of our messaging system. For example, the mapping relation from context *Good* to feature *Standard* represents the fact that if there is a good internet connection, users will get the full standard experience of the application rather than the more restricted light experience. More specifically, for every row in the mapping table, if all the contexts in the left column of that row are activated, then all corresponding features in the right column should be activated.

In order to keep the mapping simple to use and test by developers, we model the context–feature mapping as a mere conjunction of individual mappings (i.e., "rows in the mapping table"), as opposed to using more expressive logical expressions, such as for example suggested by Acher et al. (2009). Furthermore, we only allow mapping relations from contexts to features and not from features to contexts because we believe that contexts should be the sole triggers of any adaptation. Not only does this design choice simplify the control flow of our system architecture (cf. Section 3.3), it also strengthens the different roles that contexts and features play. Contexts represent situations that trigger dynamic adaptations. Features are what changes when those situations occur.

The context–feature mapping is thus a set of *N:N* relations from a set of contexts to a set of features. The features of any individual mapping relation are all activated if and only if all the contexts in the set of contexts that trigger this set of features are activated. Conversely, as soon as a single context in the set of contexts that triggers a set of features is deactivated, all features triggered by this set of contexts get deactivated as well.

Table 1 shows an example of such a mapping, extracted from the example of Fig. 1. This example shows how the different variants of the *Notification* feature (i.e., *Mute*, *Alarm* or *Vibration*) are triggered by different situations. For example, if the end-user is *Occupied*, we expect her messaging application to remain *Mute*d.

3.2.4. Converting the context–feature mapping to propositional logic

As mentioned earlier, the features in the right column of a row in the mapping table should be activated if the corresponding contexts in the left column are activated. A naive way to convert the mapping of Table 1 to a set of Boolean formulas in propositional logic would be to translate each row into a simple equivalence relation between the left and right columns:

$$Occupied \Leftrightarrow Mute \tag{1}$$

$$Quiet \Leftrightarrow Mute \tag{2}$$

$$Normal \wedge Available \wedge AudioCard \Leftrightarrow Alarm \tag{3}$$

$$Loud \wedge Available \Leftrightarrow Vibration \tag{4}$$

But this conversion is actually incorrect. Indeed, (1) and (2) would imply the equivalence of the two contexts *Occupied* and *Quiet*. However, contexts *Quiet* and *Loud* are mutually exclusive (they are part of an alternative constraint). This means that context *Occupied* will never be active at the same time as context *Loud* under these constraints. In reality however, contexts *Occupied* and *Loud* can be active at the same time, and making it impossible would be incorrect. We therefore propose a more liberal conversion where the activated contexts meet the condition to activate the corresponding features. This shall be captured with an implication:

$$Occupied \Rightarrow Mute \tag{5}$$

$$Quiet \Rightarrow Mute \tag{6}$$

$$Normal \wedge Available \wedge AudioCard \Rightarrow Alarm \tag{7}$$

$$Loud \wedge Available \Rightarrow Vibration \tag{8}$$

However, in order to guarantee a correct mapping, one also needs to ensure that when features are activated, at least one combination of contexts for which that feature becomes active must be activated too. We do this by aggregating the different combinations of contexts that activate the same features:

$$Mute \Rightarrow Occupied \vee Quiet \tag{9}$$

$$Alarm \Rightarrow Normal \wedge AudioCard \wedge Available \tag{10}$$

$$Vibration \Rightarrow Loud \wedge Available \tag{11}$$

The complete translation of Table 1 to propositional logic is then the logical conjunction of Eqs. (5) to (11). Let us now formalise this translation. With R the number of rows in the context–feature mapping, NC_r and NF_r the number of contexts resp. features in row r, $C_{r,i}$ and $F_{r,i}$ the ith context resp. feature in row r. The first step is to define the conjunction of these contexts and features for each row r, $Context\_conj_r$ resp. $Feature\_conj_r$:

$$\forall r,\ 1 \leqslant r \leqslant R : Context\_conj_r = C_{r,1} \wedge \ldots \wedge C_{r,NC_r} \tag{12}$$

$$\forall r,\ 1 \leqslant r \leqslant R : Feature\_conj_r = F_{r,1} \wedge \ldots \wedge F_{r,NF_r} \tag{13}$$

The complete translation is the logical conjunction of two sets of formulas. The first set, corresponding to Eqs. (5) to (8) in our example, is defined as follows:

$$\forall r,\ 1 \leqslant r \leqslant R : Context\_conj_r \Rightarrow Feature\_conj_r \tag{14}$$

The second set, corresponding to Eqs. (9) to (11), is defined as follows, with F the set of all the different features f mentioned in the context–feature mapping, NR_f the number of rows where the feature f appears, and $row_{f,i}$ the ith row where the feature f appears:

$$\forall f \in F : f \Rightarrow Context\_conj_{row_{f,0}} \vee \ldots \vee Context\_conj_{row_{f,NR_f}} \tag{15}$$

3.3. System architecture

Having introduced the core building blocks of our FBCOP approach, we will now elaborate upon the underlying system architecture that ties everything together. We will illustrate the

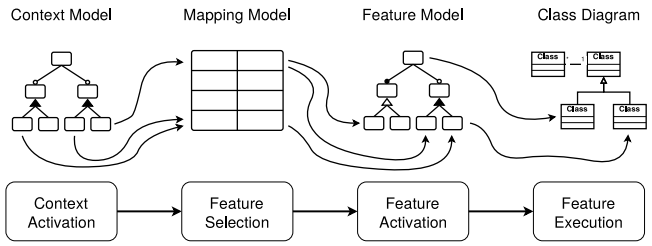

Fig. 2. Overview of the system architecture for FBCOP (Duhoux et al., 2019a).

control flow of our architecture from a high-level perspective only, using our running example. For a more implementation-oriented explanation of the system architecture we refer to our earlier implementation-oriented paper (Duhoux et al., 2019a).

Our approach revolves around our context-oriented software architecture (Mens et al., 2016) and Hartmann and Trew's *multiple-product-line-feature model* (Hartmann and Trew, 2008). In essence, our architecture and this model propose to manage and represent both the contexts to which the system adapts and the features it adapts in terms of separate independent models, as we explained in the previous section. Fig. 2 presents our overall system architecture.

The control flow of this system architecture goes as follows: whenever a change is detected in the system's surrounding environment, either by user interactions or information received from external or internal sensors, the contextual information is reified as context objects. The Context Activation component activates or deactivates reified contexts in the configuration of the context model, while respecting the constraints imposed by that model. If the updated configuration does not satisfy the context model's constraints the system rolls back to a previous valid configuration. Otherwise, the updated configuration is kept, based on which the Feature Selection component (un)selects the appropriate features thanks to the context–feature mapping. The feature Activation component then attempts to (de)activate these features in the configuration of the *feature model*, while ensuring that all constraints imposed by that model remain satisfied. Again, if the updated configuration does not satisfy the feature model's constraints, the system rolls back to a previous valid configuration. Finally when the features have been (de)activated, the Feature Execution component installs or removes the code of these features in the system to refine and adapt its behaviour while the system is running. Note that, whereas in the current state of the FBCOP architecture implementation, trying to update a system to an invalid state leads to rollbacks, in future work, alternative mechanisms such as statistically checkable properties could be used to improve the control flow between the different architectural components.

Let us illustrate the current control flow on a possible usage scenario of our messaging system. Assume an *Adult* has our messaging application on her smartphone. As she is presenting her work at a conference via her *Smartphone*, where the audience is being quiet to listen to her, the Context Activation component activates the reified contexts *Adult*, *Smartphone*, *Occupied* and *Quiet*. This triggers the Feature Selection component to select the features *Mute*, *Search*, *Description*, *Minimalist* and *Keyboard*. After verifying that they lead to a consistent configuration, these features are then activated with the Feature Activation component and finally deployed in the application by the *Feature Execution* component. Since her smartphone is now muted, when she receives new messages in the middle of her presentation, she will not be disturbed by them. After her presentation, she joins the attendees at the coffee break to discuss with them. No longer occupied, she sets her phone to available which will activate the *Available* context and deactivate the *Occupied* context. This triggers the system to deactivate the feature *Mute* and activate the feature *Alarm*, since the noise level is *Normal* during the coffee break and her smartphone has an *AudioCard*. The system then activates and deploys the feature *Alarm*, so that the user can be notified with a sound whenever she receives a new message. At the end of the coffee break, she wants to stay in touch with some people and adds them to a group chat. To do so, she *Search*es each person by name to add them to the group with the *Search* feature. This concludes our usage scenario. Wouldn't it be great to have a testing approach that could generate such scenarios?

3.4. Development methodology

Due to their high run-time adaptivity in terms of active contexts and features, conceiving FBCOP systems is hard. In addition to dedicated visualisation and testing tools, developers of such systems can benefit from a supporting methodology to put them in the right mindset and tell them what to focus on in each step of the life-cycle. This section presents such a FBCOP-specific methodology, which we suggested developers to follow during the validation experiment described in Section 7. Explaining this methodology also allows us to highlight to the reader where and how in the development life-cycle the testing approach proposed in this paper fits in.

At a high level, our methodology, summarised in Fig. 3, is a typical iterative development process consisting of four phases: *Requirements* analysis, *Design*, *Implementation* and *Testing*.

3.4.1. Requirements

During the *requirements* analysis phase, by carefully analysing the domain of the application, developers are asked to come up with a list of features of the system and a list of contexts to which the system must adapt or refine its behaviour. To avoid misunderstandings and misinterpretations, a lexicon needs to be defined containing a description of each feature and context, as well as a rationale to justify the less trivial features and contexts.

To get started, our methodology suggests to identify the main features of the system first. Two main features of our case study are *Sending* and *Receiving* messages. Using these features as starting point, developers are suggested to think about possible variants of these features and in what contexts these particular variants would be triggered. For example, developers could identify that the kind of messages that can be sent or received could depend on certain contexts, such as having a *Good* internet connection and having a *VideoCard* on the device, when including pictures in the messages.

These contexts can then be used as a next step to think about what other features or variants of existing features may be relevant to those or related contexts. Having thought about how the peripheral cards on a device can influence what type of messages can be handled, developers could identify that the presence of an *AudioCard* is required for being able to handle *Vocal* messages.

This process is repeated until the developer is happy with the set of features and contexts he has identified to include in a first version of the system to implement.

3.4.2. Design

During the design phase, the developers organise the contexts and features identified during the previous phase in a context and feature model, respectively. Additionally, in the context–feature mapping they need to declare explicitly what contexts trigger what features. For the non-trivial relations in this mapping it is advised to provide a rationale to justify their purpose. Putting all these models together yields an overall model such as the one illustrated in Fig. 1. What still needs to be added to this model is what are the main classes of the software system, in terms of a class diagram, and what features will adapt what classes.

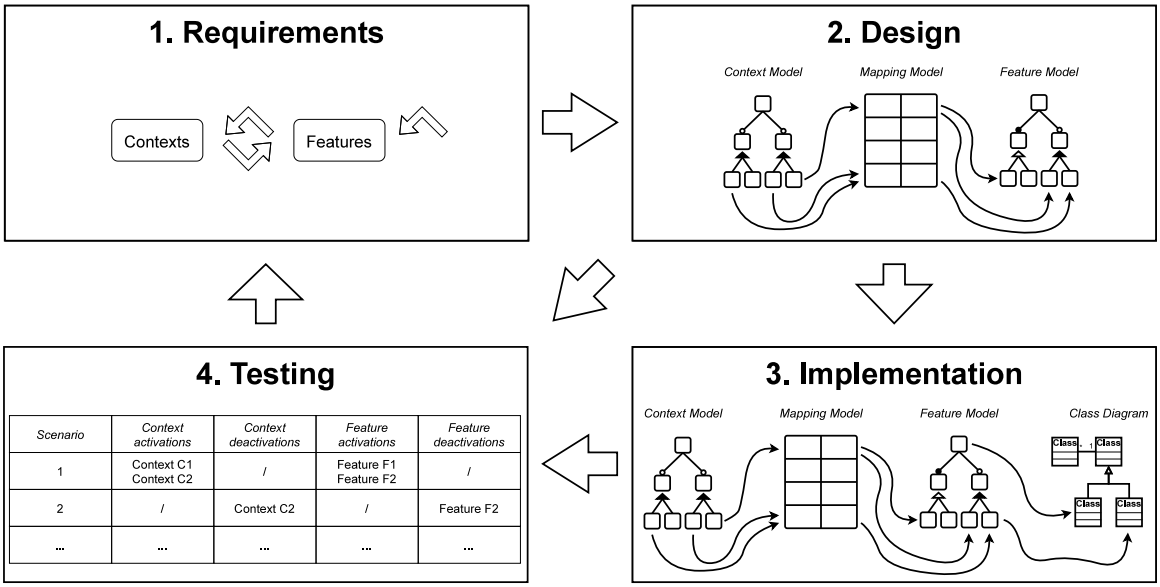

Fig. 3. Overview of our FBCOP development methodology.

3.4.3. Implementation

Now the developers can finally start to implement their FBCOP application. More information on how to do so can be found in our implementation-oriented paper (Duhoux et al., 2019a). The most important things to be implemented during this phase are the base classes and their class hierarchy, as well as the individual features that will modify or add methods defined on these classes or on previously activated features. During this implementation phase, to help developers understand or inspect the behaviour of the system they are implementing, they can make use of two supporting visualisation tools (Duhoux et al., 2018, 2019b). These tools enable a developer to visualise at runtime the activation state of the contexts and features, the dependencies between them, as well as the interaction between and within the components of their system (classes, etc.).

At the end of this phase, an executable feature-based context-oriented system is delivered.

3.4.4. Testing

When testing software product lines (Lee et al., 2012) and adaptive software systems, many aspects can be subject to testing, ranging from the requirements to the implementation of such systems. The main purpose of the testing approach proposed in this paper is to find possible faults in either the design or implementation of the system being developed. In our proposed methodology, this test phase can be carried out either immediately after the design phase, after the implementation phase or both.

Our testing approach generates relevant test scenarios that can be used to verify whether the system exhibits the expected behaviour or not. But even without simulating or executing these scenarios, they often provide useful insights in the quality of the design models. For example, if a scenario suggest to activate certain combinations of contexts or features in ways that were not intended, this may be an indication that the design is flawed. These could then be corrected even before the implementation phase has started. Sections 4.3 and 7.1 will illustrate such design testing.

Alternatively, the generated scenarios can be used after the implementation phase, to simulate and verify the correct behaviour of the system under these scenarios. Obviously, the testing approach proposed in this paper can and should be complemented with more traditional testing techniques such as unit testing.

3.4.5. Iterative methodology

Our proposed methodology is iterative. After having done a first iteration through the four phases from requirements to testing, via design and implementation, another iteration can be done to extend the first version of the system with additional contexts and features. This also implies that the test suites generated earlier need to evolve to include these new contexts and features. This question of test suite evolution will be discussed in Section 8.

4. Testing

Having introduced the main concepts of feature-based context-oriented software development, its underlying architecture and development methodology, we now revisit the five research questions that steered our research towards a testing approach for FBCOP systems, based on the automatic generation of test scenarios.

4.1. Test suite generation

Because of the large number of possible combinations of contexts and features, and the dynamic activation and deactivation of the contexts that trigger these features, exhaustively testing all possible scenarios induced by a FBCOP system becomes intractable. Our first research question (RQ1) is thus: **How to generate a pertinent yet tractable set of test scenarios for a given FBCOP application?** Each such test scenario is a configuration of the context and feature models, representing a set of situations in the environment together with a set of corresponding features to activate.

For simplicity's sake, assume that we have a feature-based context-oriented application with these three contexts ***Quiet***, ***Normal*** and ***Loud*** and two features *Photo* and *Alarm*.

Table 2 lists a subset of the 2^5 possible combinations of the activation and deactivation of these contexts and features. For clarity, we keep the contexts on the left and the features on the right. Each line is a configuration which specify the state (Activated or Deactivated) of the three contexts and two features.

Some of these scenarios may be invalid when they do not respect the constraints imposed by the context model, feature model or mapping model. E.g., given the *alternative* constraint in the context model between the contexts ***Quiet***, ***Normal*** and

Table 2
Example of an (invalid) test suite, where *A* and *D* mean that a particular context or feature is *A*ctivated, resp. *D*eactivated.

Scenario	Contexts			Features	
	Quiet	**Normal**	**Loud**	Alarm	Photo
1	D	A	D	A	A
2	A	D	D	A	D
3	D	A	A	D	D
…			…		

Loud, only one of them can be active at a time, making the third scenario invalid. The second scenario is invalid since the mapping model does not permit the feature *Alarm* to be active in context ***Quiet***.

Obviously, we want our testing approach to generate only valid scenarios, which is a main challenge. Due to the high number of constraints in feature-based context-oriented applications, random sampling would result in practically only invalid configurations. Considering the 47 contexts and features of our case study presented in Section 2, there are more than $2^{47} = 1.4e14$ possible configurations, while the number of *valid* configurations is only in the hundreds. Using random sampling, up to 1.4e12 tries may be needed before getting even one valid configuration, and there are no guarantees that the valid configurations found this way are pertinent. This problem is well-known, and techniques exist to solve it.

Hence, in Section 5, we will address RQ1 by reducing a FBCOP system to a highly reconfigurable system in presence of constraints, in order to adopt a pairwise testing approach (Cohen et al., 2008). Such computation exploits a SAT-based representation unifying the context, mapping and feature models of a FBCOP system, effectively handling the three layers of constraints present in these systems. Moreover, this approach can take advantage of particular properties of FBCOP models, as will be shown in Section 5.3 on complexity, and in Section 5.4 which will highlight the efficiency of two well-known optimisations on FBCOP systems.

To our knowledge, in the area of context-oriented programming languages, currently little research exists on the topic of testing. Whereas in this paper we mainly rely on Cohen's approach (Cohen et al., 2008), in the future we will further improve our approach by taking insights from pairwise testing for software product lines (Perrouin et al., 2012) and more diverse strategies for testing products in software product lines (do Carmo Machado et al., 2012).

4.2. Creation cost of a suite

The algorithm which we built in response to RQ1 will only generate sets of scenarios that satisfy the constraints imposed by the context, feature and mapping models. For the sake of the example, however, let us assume for now that the three scenarios shown in Table 2 are valid, for example, because the models are still an early draft that do not include all constraints yet. This assumption will allow us to illustrate later on how the testing approach can discover certain design anomalies that could help correcting or completing the models.

The presentation of scenarios as shown in Table 2 may quickly become unreadable, even for small systems with only a few tens of contexts and features. A solution to this problem is to show only the activation switches, i.e. only the differences in the contexts' and features' states, of contexts and features between subsequent scenarios instead. This idea is illustrated in Table 3.

In addition to being more compact, this representation is closer to how FBCOP systems actually work, by dynamically activating and deactivating contexts and features. This representation tells a developer exactly which context switches need to be simulated in the different test scenarios. Limiting the number of context switches, which we will call the *creation cost* of a test suite, is of key importance if one wants to reduce simulation costs, which leads us to the second research question (RQ2): **How to minimise the effort of simulating these generated test scenarios?**

Table 3
Example of a test suite with activation switches (contexts in bold).

Scenario	Activation	Deactivation
1	**Normal**, Alarm, Photo	
2	**Quiet**	**Normal**, Photo
3	**Normal**, **Loud**	**Quiet**, Alarm

Table 4
Rearranged version of the example test suite.

Scenario	Activations	Deactivations
2	**Quiet**, Alarm	
1	**Normal**, Photo	**Quiet**
3	**Loud**	Alarm, Photo

First, we observe that the particular ordering of scenarios in a test suite may affect the number of context (de)activation switches. E.g., the example of Table 3 has 6 context switches (in bold). By swapping scenarios 1 and 2, the creation cost (i.e., number of context switches in bold) would get reduced to 4, as illustrated in Table 4.

In Section 6 we will generalise this observation into a lightweight test suite rearrangement algorithm, based on the creation cost, a metric tailored to FBCOP. We define *test rearrangement* as follows (Catal and Mishra, 2013): given a test suite T, a set PT of permutations of T, and a function f from PT to the real numbers, the problem of rearranging test suite T is to find a new test suite $T' \in PT$ such that:

$$\forall T'' \in PT : T'' \neq T' \implies f\left(T'\right) \leqslant f\left(T''\right)$$

where the function f evaluates the creation cost of a test suite.

Other rearrangements were studied previously in software product lines, such as using statistical prioritisation (Devroey et al., 2017), graph representation and heuristics minimising the analysis effort (Lity et al., 2017), or similarity-based product prioritisation improving feature interaction coverage as fast as possible during testing (Al-Hajjaji et al., 2019). Test suite prioritisation has also been studied previously for software development. The cost induced by a test suite was considered, for example by analysing the length of the test configurations in the suite (Bryce et al., 2011), or the time cost of each test configuration (Srikanth et al., 2009).

4.3. Using our testing approach to detect faults

In the development methodology depicted in Fig. 3, we stated that our testing approach could be carried out immediately after the *Design* phase, to find possible design anomalies in the conceived models. Alternatively, it can be performed after the *Implementation* phase, to verify that the code executes the generated scenarios as expected. This led us to our third research question, which was divided into a first general question (RQ3a): **How effective is our approach to find errors?** and another which analyses how diverse our testing approach can be (RQ3b): **What type of errors can we find using this testing approach?**

As the testing approach is a part of a development methodology, it is meant to be used in practice by developers and not only by testing experts. This led us to a fourth research question

Table 5
Example of a usage scenario corresponding to the test suite of Table 4.

Scen.	Description	Anomaly
2	When Kim wakes up for school, he sets his phone to Quiet since his little brother is still asleep. His phone alerts him of a new message with an Alarm.	His smartphone shouldn't be using an Alarm if the mode is set to Quiet. The context–feature **mapping** may be **faulty**.
1	As Kim walks to school he switches his phone's sound level to Normal, and takes some Photos of cats on the way.	
3	Detecting a Loud ambient noise level at the tram station, Kim's smartphone turns off the Alarm feature since Kim is unlikely to hear it with all that noise.	The phone automatically activated Loud mode but did not turn off Normal mode, which is inconsistent with the requirement that only one Noise level can be active simultaneously. The context model might contain a **relationship error**.

(RQ4): **How relevant and easy-to-use is the testing approach for developers?**

To answer these research questions RQ3 and RQ4 we grouped them, since the only way to answer them is by validating them in practice when building a real context-aware application. We asked 19 pairs of developers (experienced university-level students following a master-level course on software maintenance and evolution) to design, develop and test a small FBCOP application using our approach, over the course of two months. The case that was assigned to them was the smart messaging system described in Section 2.

We asked them to use our language, tools and methodology to design, implement and test their application. After having analysed what features and contexts are required by the application, each pair of developers had to design a context–feature model such as the one depicted in Fig. 1, and then implement the actual application according to this model. This model or the implementation might still contain some errors, some of which hopefully would get detected thanks to our testing approach.

It was left open to the developers when and how to apply our testing approach, either after the design phase or after the implementation phase or both. To do so, we gave them a tool whose input is a complete context–feature model (context model, feature model, mapping model) and the output is a (rearranged) set of scenarios, such as the one in Table 4. They were suggested to turn this set of scenarios into a real-life usage scenario such as the one shown in Table 5. Possible anomalies in either the design or implementation of the system could then be detected by carefully analysing these scenarios and/or comparing them with the system's runtime behaviour. At the end of their development effort, they were asked to demo their system and to answer a questionnaire. Our analysis of this questionnaire will be the subject of Section 7.

As a reminder, for the sake of the example we assumed that the three scenarios in Tables 3 and 4 were valid (even though we knew they weren't). However, by analysing carefully the usage scenario of Table 5, the developers would come to the conclusion that their context, feature and mapping models may be incomplete or incorrect, as illustrated by the anomalies in the third column. They can then use this information to improve their models.

4.4. System evolution

Finally, it is inevitable that developers will continue to evolve their system. This evolution is captured by the loop in our development methodology of Fig. 3. This evolution involves the addition of novel features and contexts that would trigger the selection and activation of such features. For efficiency reasons and since existing test suites already consider all pairs of interest for the initial system design, this evolution could be done without recomputing the entire suite. This prompts the fourth research question (RQ5): **How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?**

Table 6
Example of an augmented test suite.

Scen.	Contexts			Features		
	…	Loud	*Tablet*	Alarm	Photo	*Complete*
1	…	D	*A*	A	A	*D*
2	…	D	*A*	A	D	*A*
3	…	A	*D*	D	D	*D*
…				…		

Let us illustrate this situation by augmenting the example of Table 2. Assume that we would like to add a context ***Tablet*** and corresponding feature *Complete* to use a layout that makes full use of the tablet's screen. We extend the original table by keeping the same scenarios and adding a column ***Tablet*** and *Complete* and then assigning them some activation choices (in italics) in Table 6. (Due to space limitations we replaced the original contexts ***Quiet*** and ***Normal*** and their assignments by "…".)

Unfortunately, adding these extra columns can make certain scenarios invalid. Consider the first scenario for example. If feature *Photo* is active, then according to the mapping, context ***VideoCard*** must be active too. Moreover, if both ***Tablet*** and ***VideoCard*** are active, then feature *Complete* must be active too, which is inconsistent with its assigned activation choice *D*.

The presence of constraints thus complicates the task of augmenting a previous test suite. Augmentation by random assignment risks making test scenarios invalid, and doing it manually would be quite cumbersome. What we need is an efficient and automatic way to assign an activation choice to the new contexts and features that keeps each test scenario valid. Section 8 will present such an augmentation algorithm including two different update strategies.

4.5. Summary of our testing approach

When building a FBCOP system, developers need testing to improve, debug and evolve their system. The goal of our proposed testing approach is to help them with the definition and evolution of interesting test suites. RQ1 will be addressed in Section 5. Our algorithms for rearranging a test suite (RQ2) and incrementally augmenting it (RQ5) are the subject of Sections 6 and 8, respectively. Together, the proposed algorithms form a complete process schematised in Fig. 4. This process and its associated tools will be used and evaluated by developers (RQ3 and RQ4) in Section 7.

5. Test suite generation

To answer **RQ1**, we need a test scenario generation algorithm for FBCOP. Our answer consists in reducing a FBCOP system to a

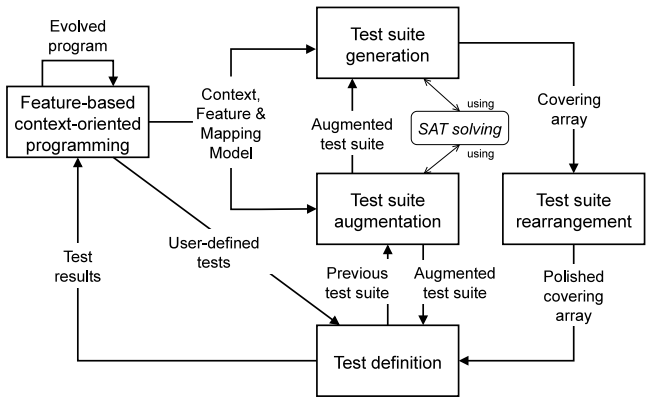

Fig. 4. Summary of our FBCOP testing methodology.

highly re-configurable system with constraints and to rely on Cohen's well-known *combinatorial interaction testing* (CIT) algorithm to generate test suites for such systems (Cohen et al., 2008).

5.1. Combinatorial interaction testing

Cohen's algorithm, which is based on a sampling methodology, takes as input a set of features with their variants (i.e., values that can be taken by a given feature and a set of constraints on such variants expressed in *conjunctive normal form* (CNF). It uses SAT solving to produce a test suite that covers any valid pair of variants.

Assume a system with k features f_i each having l_i possible values (i.e., variants). A pair of variants $((f_i, v_i), (f_j, v_j))$ is covered by a scenario when the specific value v_i for f_i (among the l_i possible values) is in the scenario along with the specific value v_j for f_j. Suppose an initially empty list of test scenarios and a list of *uncovered* valid pairs of variants (e.g., by generating all pairs, then pruning the invalid ones through SAT solving) that are not yet covered by this list of scenarios (no test scenario contains both of these variants). The algorithm works by adding new scenarios until all pairs are covered. A new scenario is selected by generating M candidate scenarios and selecting the one with highest coverage of uncovered pairs. (M is an input parameter to Cohen's algorithm which we experimentally assigned to 30 for our case study.) Each candidate scenario is constructed as follows:

(1) Find the value v (among l different values) and feature f that is present in most uncovered pairs.
(2) Let $f = f_1$, then assign randomly $f_2, \ldots, f_k$ to each remaining feature.
(3) Assume all features f_1,..., f_{n-1} have been assigned to v_1, ..., v_{n-1} and that this partial assignment respects the constraints. Now, find a value v_n for f_n (among l_n possible values), such that a maximum of pairs $((f_1, v_1), (f_n, v_n)), \ldots, ((f_{n-1}, v_{n-1}), (f_n, v_n))$ are still uncovered and that adding v_n to the partial assignment keeps it valid (checked through SAT solving).

5.2. Covering array generation

The CIT algorithm that we would like to use takes as input a set of constraints and a list of contexts and features that can be either activated or deactivated. (In other words, we consider only 0 and 1 as possible values for the variants, representing the fact that a context or feature is either active or not.) In this section, we show how to reduce the problem of test suite generation for FBCOP systems to one of test suite generation of a highly re-configurable system with constraints to which the CIT algorithm applies.

As described in Section 3 and illustrated in Fig. 1, a FBCOP system consists of a context model, a (context–feature) mapping model and feature model. Both the context and feature models are represented as *feature diagrams* (FD). Contexts or features can be assigned Boolean variables whose values represent their activation status. E.g., if we assign the value 1 (resp. 0) to the feature *MessageType*, this represents the fact that this feature is activated (resp. deactivated). Any FD can be turned into conjunctive normal form (Czarnecki and Wasowski, 2007) whose Boolean atoms are the Boolean variables corresponding to the (contexts' or) features' activation status. Any valid assignment of such a set of constraints represents the list of activated or deactivated (contexts or) features from the FD.

As explained in Section 3.2, the context–feature mapping is a set of N:N relations from contexts to features, that can straightforwardly be converted into a set of propositional logic formulas (cf. 3.2.4). The set of constraints corresponding to the overall (context, feature and mapping) model of a FBCOP system will then be the conjunction of all constraints generated by each of these three models, and the list of contexts or features to activate or deactivate for a given test scenario will be modelled by Boolean variables.

From this set of Boolean formulas in propositional logic, we can obtain a set of CNF formulas, and thereby complete our conversion of the models of a FBCOP system to a list of (contexts or) features with (Boolean) variants, in order to generate a covering array, and a set of CNF Boolean formulas, to be handled by a SAT solver.

Table 7 shows 3 of the 18 test scenarios of a generated test suite for the example presented in Section 2 and Fig. 1. The whole test suite has 18 scenarios. Scenario 13 in this Table 7, while correct, highlights a possible design flaw in our case study: it seems possible to have a *Tablet* without an appropriate *Layout*, when the *VideoCard* context is not activated. This could be solved by adding a cross-tree constraint imposing that each *Tablet* must have a *VideoCard*. Finding such design errors is precisely why we need a good testing approach.

5.3. Observations on complexity

In the covering array generation approach above, context and feature models are treated equally, in the sense that we take the conjunction of all constraints, independently of the model they represent. This means that the CIT algorithm will generate a covering array of strength 2 that covers all 2-way interactions between contexts and features alike (i.e., context–context, context–feature and feature–feature interactions).

The reason for this choice is that taking a conjunction is more efficient than trying to combine subcovering arrays that would be obtained by applying the CIT algorithm to each context/mapping / feature model separately. For example, there could exist scenarios for the covering array of the feature model for which there would be no corresponding context model configuration. We would thus have to check any combination of test scenarios for the context model with scenarios from the feature model and remove those that are not compatible with the mapping model. This removal could cause a loss of coverage, forcing us to reuse the CIT algorithm in an iterative manner and with extra constraints until reaching a fixed point. This would be a similar problem to handling constraints after computing the array, which is known to be intractable (Cohen et al., 2008).

One could argue that taking into account the interaction between contexts and features is not logical as the testing should focus on the interactions between features only. However, because contexts are always directly related to features through the

Table 7
Three test scenarios of a generated test suite.

Scenario	Context activations	Context deactivations	Feature activations	Feature deactivations
…			…	
11	Good, AudioCard, Tablet, Connection, VideoCard, Peripheral, Quiet	Normal, Smartphone	Notification, Complete, Standard, Vocal, Mode, Mute, Photo	Minimalist
12	Normal, Teen, Desktop	AudioCard, Tablet, Adult, Quiet	Match, ProfilePicture	Search, Description, Notification, Keyboard, Vocal, Mute
13	AudioCard, *Tablet*, Quiet	Good, Normal, Connection, VideoCard, Desktop	Notification, Keyboard, Vocal, Mute	Complete, *Layout*, Standard, Mode, Photo
…			…	

mapping, in practice we will generate almost no test scenarios whose goal will be to cover only pairs of contexts, or only pairs of features. Since the contexts are intrinsically related to the features through the mapping, if the models are well-designed, for a certain pair of contexts (or context–feature pair), another pair of features will always be present. E.g., the pair of context activations (*Teen*: Activated, *Good*: Activated) will always be tested with the combination of feature activations (*Match*: Activated, *Standard*: Activated), due to the mapping. If (*Teen*: Activated, *Good*: Activated) is an uncovered context pair, that also means that the feature pair is not covered either.

Another argument against our choice of not focusing on the interactions between features only, could be that adding contexts increases the number of tests. However, the size of the final covering array grows logarithmically in the number of contexts and features (Cohen et al., 1997). Let N_f be the number of features and N_c the number of contexts. Assuming that both numbers are similar, we have:

$$O(log(N_f + N_c)) \simeq O(log(2N_f)) \tag{16}$$

$$O(log(2N_f)) = O(log(2) + log(N_f)) = O(log(N_f)) \tag{17}$$

Adding contexts thus does not affect the order of complexity of the algorithm if we assume that the number of contexts and features is similar (as is the case in our case study).

In addition to what has been said above, we believe that testing pairwise contexts is also important as it forces a developer to simulate each context and make sure that they can indeed be deployed.

5.4. Computation time and optimisations

Finally, we show two optimisations of CIT algorithms that work well with the constraint model induced by FBCOP:

5.4.1. Pre-processing of core and dead contexts and features.

There may be some contexts or features that always need to remain activated or deactivated in order to satisfy the constraints. It is known that identifying such core and dead feature reduces the effort from the SAT solver. Let C (resp. D) be the number of core (resp. dead) features and contexts, M the number of candidate scenarios, and S the size of a generated covering array. Identifying core and dead features allows us to save $(C+D)\times S\times M$ calls to the solver.

The simplest way to identify a dead (resp. core) feature is to check satisfiability of a partial configuration with only this feature being activated (resp. deactivated). If this partial configuration is not satisfiable, it means that the feature must always be deactivated (resp. activated) in all configurations. Using again N_f as the number of features and N_c the number of contexts, it would cost $2 \times (N_f + N_c)$ calls to the solver.

In a concrete feature-based context-oriented application, this cost would be negligible compared to the number of avoided calls. In our case study, a low estimate of the saved calls would be 4500 ($C + D = 10$, $S = 15$, $M = 30$), while a high estimate of the cost would be 120 ($N_f + N_c = 60$). The cost is thus less than 3% of the possible gain.

5.4.2. Additional step based on propagation

An important step of SAT solvers is Boolean constraint propagation (BCP) (Marques-Silva and Sakallah, 1999). From a CNF formula and partial truth assignment (or partial configuration), the BCP phase has the objective of producing a *unit clause*, that is a clause that contains a single unbound variable. The value of that unbound variable can then be inferred, as all clauses of a CNF formula must be true for the formula to hold.

To illustrate the intuition behind this propagation, let us consider contexts *Quiet*, *Normal* and *Loud*, and the features *Mute*, *Alarm* and *Vibration*. We view these contexts and features as Boolean variables which are either true (activated) or false (deactivated). Assume that context *Quiet* has been assigned the value true. This context cannot be active if either contexts *Normal* or *Loud* are active. Hence, we can infer to assign the false value to those two contexts. We also know that context *Quiet* implies feature *Mute* via the mapping, and that *Mute* cannot be active if either feature *Alarm* or *Vibration* are active. Consequently, we can then infer to assign a true value to *Mute*, and false to *Alarm* and *Vibration*.

Cohen et al. (2008) proposed an optimisation which exploits BCP. Let $inf_{V_1}, \ldots, inf_{V_m}$ be the m values inferred by BCP, and inf_{F_k} denote the features whose value inf_{V_k} were inferred. The modification to step (3) of the CIT algorithm (described in Section 5.1) as well as the addition of fourth step (4) are as follows:

(3) Assume all features $f_1,\ldots, f_{n-1}$ have been assigned to v_1, …, v_{n-1} and that this partial assignment respects the constraints. **If f_n was already assigned an inferred value, skip this step. Otherwise,** find a value v_n for f_n (among l_n possible values), such that a maximum of pairs $((f_1, v_1), (f_n, v_n)), \ldots, ((f_{n-1}, v_{n-1}), (f_n, v_n))$ are still uncovered and that adding v_n to the partial assignment keeps it valid (checked through SAT solving).

(4) After choosing the value v_n for feature f_n, find the propagated values $inf_{V_1}, \ldots inf_{V_m}$ thanks to SAT solving and assign to each feature inf_{F_k} its corresponding value inf_{V_k}.

Table 8 shows the number of propagations and the number of propagated values with or without the first optimisation, when applied to our case study. It shows that these optimisations affect two largely separate aspects. Indeed, the number of propagated values is almost the same with or without the first optimisation. This is mainly due to the design phase as it is unlikely that a core (resp. dead) context/feature which is by definition always activated (resp. deactivated) would have an impact on other highly dynamic features/contexts of a FBCOP system.

Table 8
Impact of the pre-processing optimisation on the number of propagations and propagated values.

	Propagations	Propagated values
Without optimisation 1	12567	14282
With optimisation 1	3327	13965

Table 9
Impact of the optimisations on the computation time.

	Computation time (s)
Without optimisation	17.2
Optimisation 1	4.6
Optimisation 2	14.6
Both optimisations	**2**

Table 9 illustrates how each of these optimisations improve the average computation time for 30 executions of the CIT algorithm on our case study. Pre-processing of core/dead contexts and features drastically prunes the set of features and corresponding constraints. The first optimisation reduces the computation time by 73%, the second one decreases it by (only) 15%. The two optimisations combined reduce the time of computation by an average of 88%.

5.5. Summary

To answer RQ1, we proposed an adaptation of Cohen et al.'s CIT algorithm to generate test scenarios for FBCOP systems. An optimised version of this algorithm takes only 2 s to run on our case study, illustrating its lightweight nature.

6. Test suite rearrangement

As stated in Section 4.2, the effort for a developer to implement a generated test suite can be minimised by decreasing its creation cost. We defined the creation cost of a test suite as the number of context switches (activations or deactivations) needed to simulate all scenarios from the suite. In this section we answer **RQ2** by presenting a test suite rearrangement algorithm which minimises this cost.

6.1. Rearrangement algorithm

We define the *distance* between two test scenarios as the number of activated (resp. deactivated) contexts in the first scenario which get deactivated (resp. activated) in the second one. In other words, the distance between two test scenarios is the number of context switches needed to go from one configuration of the context model to the other. The purpose of the rearrangement algorithm is to rearrange the different scenarios in such a way that the total distance between subsequent scenarios in the test suite remains as small as possible.

The pseudo code of our rearrangement algorithm can be found in Algorithm 1. It takes a list of test scenarios L_0 and a default configuration t_0 where all contexts are deactivated which will act as initial scenario. The algorithm maintains an ordered list L that is used to build the new rearranged test suite. The algorithm uses an auxiliary method *minDistance* (L_0, *test*), whose goal is to find the test scenario in L_0 that is closest to a given scenario *test*.

As an example, the rearranged test suite shown in Table 10 has a creation cost of 89 as opposed to 161 as for the original test suite. This is a decrease of 45%. Averaged over 30 executions of the CIT algorithm on our case study, the creation cost of the rearranged test suites is 85, as opposed to 152 for the original test suites. We observed a stable decrease of 46% in cost, with a variance of 4.4% over all executions of the CIT algorithm.

Algorithm 1: Rearrangement algorithm

```
input  : L_0, t_0
output : L
1  L = new List()
2  test = t_0
3  while L_0 is not empty do
4      bestTest = minDistance(L_0, test)
5      L.add(bestTest)
6      L_0.remove(bestTest)
7      test = bestTest
8  end
9  return L
```

Table 10
Three test scenarios of a re-arranged test suite

Scen.	Context activations	Context deactivations	Feature activations	Feature deactivations
…			…	
3	Connection, Normal, Bad	Quiet	Mode, Light	Notification, Mute
4	Peripheral, AudioCard		Notification, Alarm, Vocal	
5	Smartphone, Teen	Adult, Tablet	Match, Layout, ProfilePicture, Minimalist	Description, Search
…			…	

When applying a series of tests, it is important to isolate the features or feature interactions that are causing failures. When a failure occurs in a given test scenario, it is likely that the failure is caused by (an unexpected or undesired interaction with) a new feature added in this test scenario. Although the goal of the rearrangement algorithm is to reduce the number of context switches, as an indirect side-effect it also drastically reduces the number of feature switches and hence the number of feature interactions to inspect between each test scenario. Over 30 executions of the CIT algorithm on our case study, the number of *feature* switches also decreased by 41%, on average.

6.2. Testing and order of activation

In adaptive software systems, features can adapt the behaviour of the system by being activated or not. When multiple features affect the same part of a system, unexpected feature interaction errors might occur. Thankfully, the generated test suite covers all possible pairs of features. However, a particular type of error in context-aware systems is related to the *order* in which features are activated, and this is not theoretically covered by our approach. During the testing process, we want to increase our chances to discover such critical and error-prone sequences of activations. To this end, in this subsection, we show how to obtain the feature activation order of a test scenario in a generated test suite, and discuss how our rearrangement algorithm affects the discovery of errors related to the activation order. This subsection aims at posing the foundation for future research on the activation order, e.g. with specific criteria to optimise the discovery of order-related errors, as well as further motivate the relevance of our cost-based criterion with context-aware systems.

First, let us illustrate such an error with an example. Assume that both F_1 and F_2 are features that affect a same method M, where F_2 simply refines method M but feature F_1 overwrites it with a new alternative behaviour. If F_1 is activated before F_2, it replaces M's behaviour by a new one and then F_2 further refines

Table 11
Feature activation order for the same test scenario in the unordered and reordered test suite.

Reordered	Test scenario number	Activated at test scenario					
		Activated features					
No	13	10 Notification	11 Keyboard	12 Layout	13 Mode, Photo, Search, Standard, Vibration, Complete, Description		
Yes	10	2 Layout	4 Keyboard	7 Mode	8 Notification	9 Search, Vibration, Description	10 Photo, Standard, Complete

that new behaviour, as intended. However, if F_1 gets activated after F_2, F_2 first refines the behaviour of M, but then F_1 erases that refined behaviour, which may not be what is intended, thus causing erratic behaviour.

To help developers visualise in what order features were activated for a particular test scenario, this information can easily be extracted from the test suite. A feature activated at test scenario T_i is activated on top of the features activated from test scenarios T_0 to T_{i-1}.

Let O_i represent the list of activated features in test scenario T_i, respecting their order of activation: the first feature in this list was the first to be activated, and so on. $F_{A,i}$ is the list of features activated at test scenario T_i, and $F_{D,i}$ the list of features deactivated. We can then define the following recurrence rule:

$$O_i = \begin{cases} F_{A,i} & \text{if } i = 1 \\ O_{i-1} \setminus F_{D,i} + F_{A,i} & \text{if } i > 1 \end{cases}$$

This formula defines that O_i is equal to O_{i-1}, the previous activation order, minus the features that got deactivated in test scenario T_i ($F_{D,i}$) and with the addition (at the end of the list) of the newly-activated features ($F_{A,i}$).

Table 11 shows the activation order of the same test scenario from both the unordered suite (i.e. the original order when generated as explained in Section 5) and the reordered suite. We list only the activated and non-core features of this configuration and indicate for each feature at what scenario it first got activated. (Features which are no longer active in the current scenario are not shown, and for features that got deactivated before and then activated again only their last activation is shown.) A first thing that can be observed is that for a given test scenario, the activation order changes drastically by rearranging the suite.

We now study whether such a rearrangement positively or negatively affects our chances of finding errors caused by the activation order. Let us assume that a set of feature (de)activations was applied after already having activated a set of features in a particular order, and that a new error is detected after testing the new activation order. Locating the cause of this error will be more easy if (1) the original activation order was thoroughly tested previously and is therefore not likely to be the cause of the error and (2) the applied modification (i.e. the set of feature (de)activations) is small, helping us to isolate the actual changes in the system and what change caused the error. Moreover, more errors can be found if (3) more different activation orders are tested throughout the test suite (coverage criterion).

6.2.1. Testing score of an activation order

Intuitively, we could say that an activation order O_i is "thoroughly tested" if it went through multiple testing phases, or if multiple parts of this order were already tested previously. In the context of a test suite, parts of this activation order were tested previously if they subsisted through several test scenarios and were already used in previous tests.

This is illustrated in Table 11, where we track for all features active in a given test scenario, in which earlier scenario they actually got activated. In the case of test scenario 13 from the unordered test suite, we can observe that the feature *Notification* already got activated in scenario 10, and was subsequently tested together with feature *Keyboard* in test scenario 11. The activation order *Notification*, *Keyboard*, *Layout* was then tested in scenario 12. In other words, several parts of the entire feature activation order of test scenario 13 were already tested repeatedly before that scenario. The more parts and the more they were tested before, the better it is.

To analyse whether our rearrangement has a positive impact, we propose a simple metric: the *testing order score* $O_{score,i}$ of an activation order O_i is the number of test scenarios $O_{activation,i}$ involved in the activation of all features in O_i (including scenario i), normalised by the total number of features activated in this test scenario $N_{f,i}$:

$$O_{score,i} = \frac{O_{activation,i}}{N_{f,i}}$$

The score's value is comprised between 0 and 1; a score closer to 0 means that there are less prior test scenarios, a score of 1 means that there are as many prior test scenarios as there are features activated in the final test scenario.

On our example, the testing order score for scenario 13 of the unordered test suite is 4/10 = 0.4 (counting the 4 scenarios 10, 11, 12 and 13), while the score of the corresponding scenario 10 for the reordered test suite is 6/10 = 0.6 (counting scenarios 2, 4, 7, 8, 9 and 10). Parts of the features involved in the final scenario were thus tested more in the reordered case than in the unordered case.

We can generalise this score to an entire test suite by averaging the testing order score over all test scenarios in the suite, from scenario T_5 to T_n, where n is the last test scenario. We exclude the test scenarios T_i with $i < 5$ from the average, since these early scenarios tend to have a low testing order score simply because they have less prior test scenarios to profit from. On our case study, the average score of 30 unordered and their corresponding reordered test suites is respectively 0.37 versus 0.55. We thus observe an improvement of 47% in the testing order score for the rearranged suite.

6.2.2. Size of the modification

The second criterion is whether the applied modifications are small or not. For example, the last modification for the unordered scenario 13 in Table 11 activates 7 features at the same time, as opposed to the reordered scenario, where the last 7 feature activations are spread over 3 different modifications. On average, a consequence of our rearrangement algorithm is that the number of feature switches is decreased by 41% as compared to the unordered case. Consequently, $F_{A,i}$ and $F_{D,i}$ will also be smaller by 41%, on average. From this result and the previous one, we argue that our rearrangement makes it easier to find errors caused by the activation order.

6.2.3. Coverage of activation orders

Our third criterion was the coverage criterion which states that the more different activation orders are tested throughout the test suite, the more likely we are to find errors due to the activation order. To study the effect of our rearrangement algorithm on this coverage, we define that an activation order covers the ordered tuple $[F1, F2]$ (i.e. this pair is different from

[$F2, F1$]) if the feature *F1* was activated before *F2*. The *coverage score* of a test suite is then the total number of different ordered pairs covered through all *n* activation orders of its *n* test scenarios, divided by the total number of possible ordered pairs of features. The more pairs are covered, the better the coverage score. This score ranges from 0 to 1, where 0 means that the test suite covers no pairs of features (i.e. the test suite is empty) and 1 means that it covers them all.

Our case study contains 252 ordered pairs of features. With 30 generated test suites, on average the unordered test suites covers 172 pairs, while the reordered test suites covers 182 pairs. Our rearrangement algorithm thus improves the coverage score by a stable 2.5%. So while our rearrangement does not improve the coverage of activation orders by a lot, at least it does not deteriorate it.

6.2.4. Impact of rearrangement on testing the order of activation

In summary, even though the prime goal of rearrangement is to reduce the simulation effort, it also has a positive effect on our ability to locate errors caused by the activation order, thanks to better tested activation orders and smaller modifications. Moreover, even though the rearranged test scenarios are more compact, this does not seem to impact our coverage of activation orders. We thus believe that our rearrangement algorithm based on cost-based prioritisation is particularly interesting for implementation testing purposes: it reduces the overall simulation effort for testers and increases their chances for locating feature and feature interaction errors more easily.

Future work could explore a criterion that directly takes the order between features activation into account, and study the trade-off on the cost.

6.3. Comparison to SPL rearrangement

Feature-based context-oriented programming reuses ideas from software product lines, notably their feature modelling. This similarity triggered us to reuse previous work on combinatorial interaction testing to generate our test suites. Other testing concerns are relevant for both FBCOP and SPL as well.

SPL testing often suffers from limited test resources which can lead to the testing process being abruptly stopped. Early fault detection, i.e. detecting faults as fast as possible, is thus encouraged and sought after and is the goal of most prioritisation techniques in SPL test prioritisation literature (Catal and Mishra, 2013).

Similarly, this section focused on optimising test resources for FBCOP systems. However, we focused on a different criterion to prioritise and rearrange the tests, namely the creation cost (as opposed to the cost of executing the tests). Indeed, we assume that testers will use the entirety (or a majority) of a generated test suite. To reduce the overall cost of creating the suite we leveraged the clear separation of contexts and features. On top of that, we observed how the rearranged suite makes errors easier to locate for FBCOP systems. Hence, we believe this alternative creation cost-based approach is better suited for FBCOP.

In this subsection, we further justify our need for this alternative prioritisation approach by comparing it to previously studied criteria used in SPL testing (Sánchez et al., 2014). To achieve early fault detection, Sanchez et al. discuss four effective prioritisation criteria (a fifth, based on the commonality degree of features, proved to perform badly, so we won't consider it here).

The first 3 criteria were based on the complexity of the feature model and prioritise configurations that are more complex. This because more complex configurations are assumed to be more error-prone and therefore should be tested before less complex ones.

A typical metric of feature model complexity is the Cross-Tree Constraints Ratio (CTCR). The CTCR of a feature model is the ratio of the number of features (repeated features counted once) in the cross-tree constraints (that are typically inclusion or exclusion constraints) to the total number of features in the model (Bagheri and Gasevic, 2011; Benavides et al., 2010). For all case studies considered by Sánchez et al. (2014), their CTCR was between 0 and 25%, and they used both randomly generated feature models and models from the SPLOT repository (Mendonca et al., 2009).

The spirit of this metric is to evaluate the number of extra constraints on the features, i.e. the constraints that are not related to hierarchic relationships within the tree. Transposing this to our context, feature and context–feature mapping models for FBCOP, this means that we should consider all relationships within the mapping model to be extra constraints on the features. Doing that would yield a CTRC of 46% (13 features over 28 total features) for our case study, surpassing by almost a factor 2 those of Sanchez et al.'s study, and more complex context–feature mappings could yield even higher scores.

Given the nature of how FBCOP systems are modelled, this metric seems to lose its pertinence when applied to such systems. It is used to differentiate features involved in cross-tree constraints from those that are not; i.e., the features that are potentially more complex and more error-prone. In a FBCOP system, however, nearly all features need to be activated by at least one relationship within the mapping model in order to be relevant.

Similar problems hold for other feature model complexity metrics. They do not seem to transpose well to the case of FBCOP systems. Because of the different way context–feature models are structured, they require different metrics.

The last criterion proposed by Sanchez et al. is a dissimilarity prioritisation criterion. It assumes that a set of dissimilar test scenarios has a higher fault detection than a set of similar ones. The former are more likely to cover more components than the latter. With this criterion, the most dissimilar test scenarios will be tested first.

In order to adapt this criterion to FBCOP systems, we observe that its goal is to test dissimilar behaviours of the system under test. Hence, in the context of FBCOP systems, we could define it as the number of feature switches (i.e. activations or deactivations of features) between two test scenarios (as features, unlike contexts, capture the behavioural variants of the system under test).

However, while we proposed to minimise the number of context switches in order to reduce the simulation cost, dissimilarity prioritisation can be seen as maximising the number of feature switches. These two goals are incompatible. As we saw, an indirect consequence of our cost-based criterion was that the number of feature switches got reduced by 41% as well. Trying to combine the two seems impossible: the number of context switches and the number of feature switches between two test scenarios, although not equal, are expected to be proportional in most cases.

Previous work already studied both cost-based prioritisation and early fault detection, but with different modelling techniques (Srikanth et al., 2009). Given their opposed tendency it remains to be seen if it would be worthwhile considering hybrid criteria that reconcile both cost reduction and early fault detection at once. Once such hybrid criteria are developed, we would be able to evaluate quantitatively the trade-off between creation cost and early fault detection. For now, we chose the path of focusing on cost reduction exclusively, which seemed to make more sense for FBCOP systems.

Instead of trying to combine our cost-based criterion with one focusing on early fault detection, our criterion could be improved in other ways. For example, until now, we simply considered

the value of a context switch to be unary. Instead, different weights could be assigned to contexts that take a longer time to be switched (e.g., a context activating a feature with a long preparation time to become operational). With more data on the operation of FBCOP systems, we could take into account different measurements on the system to further reduce the simulation effort, by defining such weights.

6.4. Summary

Our answer to RQ2 was a simple-to-implement rearrangement algorithm which greedily tries to minimise the creation cost of a test suite, a metric closely related to the effort of simulating and creating this test suite. We also discussed how the rearrangement has a positive effect on a tester's ability to detect errors caused by unexpected feature interactions. Finally, we discussed what distinguishes the rearrangement of test suites for FBCOP systems from SPL test prioritisation.

7. Validation

In Section 4.3, we mentioned two ways in which a developer could use the generated test suite. As suggested by our development methodology of Section 3.4, we can use it to test not only the implementation of a FBCOP system, but also its design. To answer **RQ3** and **RQ4**, we will assess the effectiveness, relevance and usability of our testing approach with an actual context-aware application. Hence, in Section 7.1, we illustrate on our case study of a smart messaging system some errors in the design of the context model, feature model or context–feature mapping that could be discovered when applying our testing approach. This qualitative study specifically answers to **RQ3a** and **RQ3b**. Next, in Section 7.2 we analyse the results of an experiment where 19 pairs of developers adopted our development methodology and used our testing approach to build such a system. This larger experiment complements our qualitative study, as well as evaluates the relevance and ease-of-use of our testing approach with developers (**RQ4**).

7.1. Design errors

To illustrate the process of design testing on our case study, we generated a test suite using the CIT algorithm presented in Section 5 and rearranged it to reduce its creation cost, hopefully obtaining a pertinent yet light-weight test suite. This test suite was then used as a guideline to write a set of real-life usage scenarios, just like we explained in Section 4.3. The results are shown in Table 12.

For the sake of the example, we only show the most interesting usage scenarios and the possible design problems they may evoke. The complete version, as well as the generated test suite it is based on, is available on a public repository: https://github.com/PierreMartou/SPLC-Special-Issue-Annexes.

Our analysis of possible design problems in these usage scenarios yielded interesting results. It highlighted several design issues, such as two errors in the context–feature mapping, an incoherent feature adaptation and an incorrect modelling of the environment. We selected these parts of our entire analysis on purpose, as they illustrate design problems and their solutions in each of the different models (feature model, context model, context–feature mapping). Line 12 of Table 12 also shows how the writing and analysis of usage scenarios can serve as a source of inspiration for future improvements of the application. This is especially important since evolution is a key part of our proposed development methodology (cf. Section 3.4).

Lastly, notice how this usage scenario is composed of relatively short tests. Short usage scenarios should be sought after, as they are easier to write and analyse. The length of a usage scenario is naturally linked to the number of context and feature switches of its corresponding test scenario. Hence, reducing the number of context and feature switches also reduces the time for writing and analysing usage scenarios.

At the end of Section 6.2, we argued that our cost-based prioritisation and criterion were useful for implementation testing purposes. The results of this Section 7.1 show that they are also relevant for design testing purposes.

With this experience, we gained confidence that our testing approach is indeed effective for design testing, at least on this case study. A more general study will be the subject of the next subsection, where 19 pairs of students playing the roles of designers and developers were asked to use our testing approach.

7.2. Experiment

Since FBCOP is still relatively novel, it lacks industrial case studies and data on errors in these cases (Sánchez et al., 2014). We therefore opted for a more qualitative experiment to validate the approach. As future work, we could study how to generate random but pertinent context and feature models as well as a tailored fault seeding, in order to conduct a more quantitative study.

7.2.1. Set-up

Our qualitative study was conducted with 19 pairs of developers, as announced in Section 4.3. Each pair designed, developed and tested a small FBCOP application, over the course of two months. To this end, they used our FBCOP software development language, tools and methodology, which were detailed in Section 3. The case assigned to them was the smart messaging system introduced in Section 2. On average, their context–feature models comprised 20 contexts, 28 features, 15 mapping relations and generated test suites of size 12.

As suggested in Section 3.4.4, they used our testing approach both after designing their application and after implementing it. For design testing, they followed the same steps as in 7.1, by first turning the generated set of scenarios into real-life usage scenarios and then analysing these. For implementation testing, they simulated the generated test suite directly on the system and verified whether the system behaved as expected.

The experiment took place during a software engineering course taught to computer science and engineering master students at university. The students had good programming background, with active knowledge of several programming languages (Python, Java, C,) and having participated in several programming projects. For this reason, we believe they were qualified enough to play the role of designers and developers to evaluate our testing approach.

Though experimented programmers, they had no knowledge of context-oriented programming, so they could rely only on the provided guidance and documents on our development and testing approach but not on any prior knowledge to develop their context-aware application. Eager to learn, they did not have negative bias due to reticence to move away from a more familiar methodology, nor did they have positive bias because of prior experience helping them to understand and design such systems.

At the end of their development effort, the students were asked to demo their system and to answer a questionnaire anonymously. The full version of this questionnaire including the answers is available on the following public repository: https://github.com/PierreMartou/SPLC-Special-Issue-Annexes. This repository also includes the executable tool the participants used,

Table 12
Real-life usage scenario of our case study, based on a generated test suite, with an analysis of possible design problems.

N	Scenario description	Analysis of possible problems
	Situation: a group of friends discuss with each other through the messaging application.	
1	Malcolm (an Adult) is messaging his friends on his Tablet at a Loud coffee shop. His Tablet vibrates when he receives a message from one of his friends, Alma (another Adult).	
2	Alma receives a message back from her friend Malcolm and replies with her Tablet using 4G, while walking on the street.	*How was Alma notified of the message from Malcolm, if there was no Notification feature active? There might be **a mapping error**. Mute should be activated by default if neither the condition to activate Alarm nor Vibration are met.*
3	…	
4	Likewise, Danilo (an Adult) was at home when he was alerted of a message. With a simple glance at his cutting edge Tablet, he enjoys a Complete Layout and immediately sees the complete information: the message, the hour, the group of friends.	*In this step of the scenario, the features Layout and Complete were activated. But what was the Layout then in the previous steps since no Layout was explicitly activated there. Does it make sense for the messaging application to have no Layout? There seems to be an **incoherent feature adaptation** of the system to the contexts. The feature model lacks a default feature or an alternative to having an explicit Layout.*
	Situation: a group of teenagers discuss with each other through the messaging application.	
5	Fallon (a Teenager) is in a park, where she has no Connection. She writes messages on her Smartphone using the virtual Keyboard. She's also taking selfies to update her Profile Picture with the camera of her Smartphone.	*We have no information on her Connection, yet Sending messages is still active. She shouldn't be able to update photos or send messages if there is no Connection. The context model lacks information and the environment is **incorrectly modelled**.*
	…	
	Situation: several people use a Desktop computer in a cybercafé, where there is a lot of ambient Noise.	
11	Ed (another Teenager) is playing on a Desktop computer in a cybercafé. He's chatting with a friend and is alerted through Vibration when his friend finally answers him.	*Since the environment is Loud, he's alerted by a Vibration, which is expected. However, it seems unexpected to receive a Vibration notification when working on a Desktop computer. There might be a **mapping error**.*
12	Not far from Ed, a man is sipping his coffee while scrolling on his cutting-edge Smartphone, sending Photos and listening to Vocal messages.	*Should he be able to listen to the Vocal messages when being in a Loud environment? While not necessarily an error, we could **improve the application** by adding TextToSpeech and SpeechToText features. By converting the spoken message to written text, the teenager could then still consult a vocal message even when in a loud environment.*
	…	

which takes as input three text files describing the models of a feature-based context-oriented system and outputs a reordered test suite, such as described in Sections 5 and 6 (cf. instructions in the README). The participants had to design a system and perform design testing in order to answer the questionnaire. Both their models (formatted in text files and correct inputs for the executable tool) and their design analysis can be found in the repository, for each group of participants.

7.2.2. Analysis

We analysed the results from this experiment from three angles: (1) the perceived *effectiveness* of our testing approach in finding errors, (2) its *completeness* (what different types of errors it covers) and (3) its *strengths and weaknesses*. Since the proposed testing approach can be used either for design or implementation testing, we also divide our discussion along these phases: Section 7.2.3 reports on design testing whereas Section 7.2.4 discusses the results for implementation testing.

7.2.3. Design testing

1. Effectiveness. We assess the perceived effectiveness of our approach to test the design, by counting the number of errors developers found by analysing the usage scenarios they wrote based on a generated test suite. We asked each developer pair to provide an estimate of how many modifications they made to their design as a consequence of their analysis. Fig. 5 shows their answers.

From these results, design testing seems pretty effective. 73.6% found at least two errors in their design models and it was useful at least once for all pairs of students but one. This pair stated in their open answers that their models were too simple, so that they could easily correct their models by hand, and that using the testing approach would have been overkill.

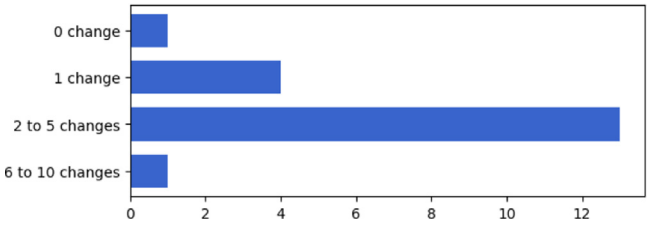

Fig. 5. Number of modifications made to the design.

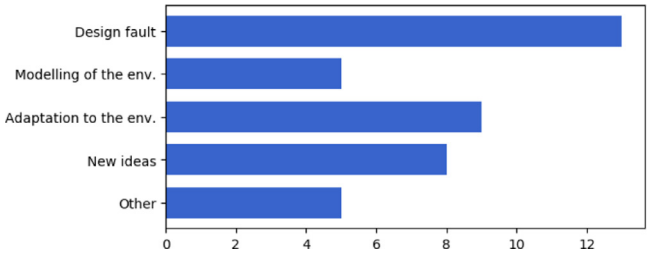

Fig. 6. Nature of modifications applied to fix the design errors. ('env.' stands for 'environment').

2. Completeness. To evaluate this criterion, we asked each developer pair for the main causes and natures of the modifications they needed to make to their design. The more types of design errors developers could find, the more complete the testing approach is. Whereas we offered some types of design errors to choose from (Fig. 6), other types could be provided as open answers.

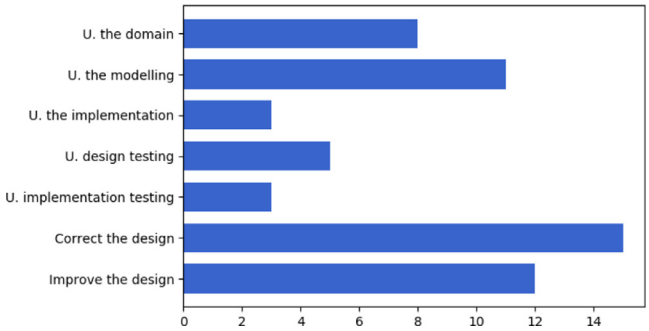

Fig. 7. Perceived strengths of using the approach for design testing ('U.' stands for 'Understanding').

The results are promising: design testing discovered varied types of errors whose causes were located in the various design models: context model, feature model, context–feature mapping. This is further justified by the open answers where the developers were asked to elaborate on the types of errors encountered. For the context model, one mentions a "modification of relationships to alternative" because the "testing scenario implied the user was in two contexts that were highly unlikely to be in at the same time". Another pair mentions they corrected the context model because it allowed for "impossible situations" in the test scenarios. The feature model was also corrected, for example because "features were never activated". Modifications occurred in the mapping model because the "mapping wasn't coherent". The mapping having illogical implications between contexts and features (and hence the test suite including illogical situations) was the most cited cause of errors.

In addition to discovering design errors, the usage scenarios sometimes provided pertinent ideas (new contexts or features) on how to improve the design models, as shown by the "New Ideas" bar in Fig. 6.

3. Strengths and weaknesses. To assess the strengths and weaknesses of our testing approach, in addition to soliciting open responses, we also proposed of set of possibilities to choose from, to incite the participants to analyse the forces and difficulties of our testing approach from several angles.

Regarding possible strengths of our approach we suggested 7 choices as shown in Fig. 7. Five were a better understanding of the **application domain**, the **modelling**, what to **implement**, **design testing**, and **implementation testing**. The two last were whether our approach helped to either **correct** or **improve their models**.

From the responses obtained we can observe three contrasting tendencies. Firstly, our testing approach clearly helped developers to correct and improve their design, which was the primary goal.

Half of the pairs also gained a better understanding of the application domain and its modelling thanks to our approach. The main reasons for this and arguments for the effectiveness of the approach, can be understood from their open comments:

- "It's a very practical way of seeing things. Combinations that we wouldn't have come up with are generated"..
- "The testing generation tool delivers a good coverage of the possible behaviours of the system. […] this allowed us to change our models so that they represent our ideas more".
- "For the given use-case, we had between 14–16 tests which is not too much to be called overkill, but not too few to let things uncovered".
- "It is very helpful to visualise and to correct our model by detecting anomalies that we would not have thought of".

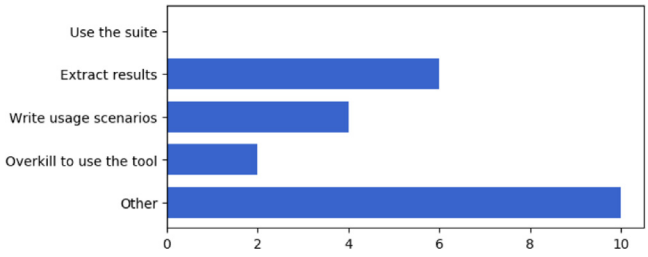

Fig. 8. Perceived weaknesses for design testing.

Pertinence and diversity of the generated test scenarios (thanks to the full pairwise coverage) are two of the main perceived reasons leading to a better understanding of their own application and modelling. It allowed them to analyse unexpected situations and confirms that an automatic generation of test scenarios using pairwise testing is useful to guide the analysis of a context-aware design.

Only a minority considered the approach helpful to better understand their own implementation or how to test it. This is possibly due to a bias because we provided more guidance on how to use our testing approach for design testing, as opposed to how to use it to write implementation tests.

We assessed the weaknesses in a similar way. The 4 proposed choices were: they didn't understand **how to use the generated test suite**, it was too hard to **extract interesting results** from the usage scenarios, the generated test suite didn't help to **write realistic usage scenarios**, and **using the tool was overkill** for example because their models were too simple. The results are shown in Fig. 8.

Since no one chose the first answer, we can confirm that all groups understood how to use the tool and what was expected of them. The two pairs who chose the answer that it was "overkill to use the tool" also believed that writing usage scenarios and extracting results was difficult. Although they do not provide reasons for this in their open responses, it seems logical that finding errors in simple models is difficult, because there could be none.

Half of the remaining comments were about tool usage (e.g., formatting of input and output), rather than about the results themselves. Indeed, being a first prototype, the tool still used a rudimentary presentation and had to be run as stand-alone tool not integrated with the rest of the programming framework. This can be resolved in future experiments by using a more complete tool which provides better integration, a friendlier user interface, and support to save and restore generated test suites to avoid inadvertently losing an interesting test suite while working on it (as the generation is random).

Another issue is illustrated by this comment: "Scenarios were created that had no real life 'equivalent'. It shed a light on our problems but we had difficulties resolving them". As mentioned, our study participants were novices in designing context-aware systems. We believe that some of them suffered from their inexperience and therefore designed unrealistic applications at first. This weakness could be diminished by providing more beginner-friendly resources on FBCOP (more documentation, examples, tutorials, cases).

7.2.4. Implementation testing

After having designed their application, each pair of students started to implement an actual prototype, testing it manually or with unit tests, until basic usage was possible. Only then they generated and simulated a test suite using our tool. This part of our study concerns only their use of this suite (i.e. not any of

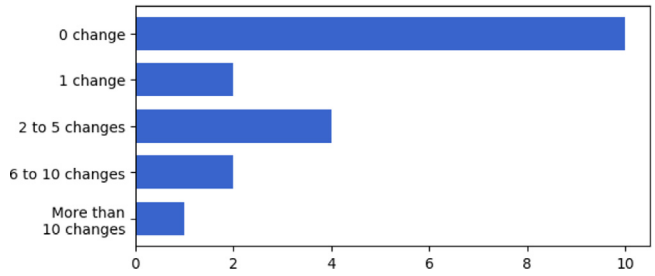

Fig. 9. Number of modifications to the implementation.

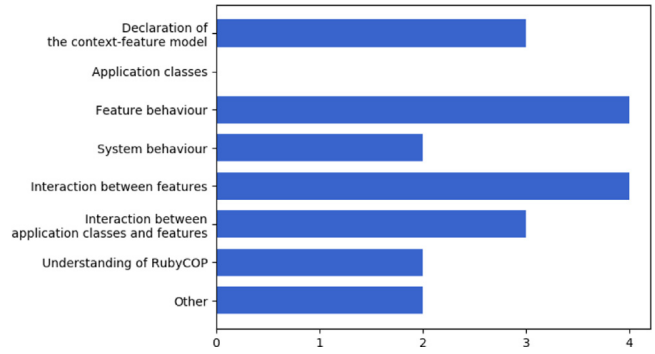

Fig. 10. Possible causes of errors in the implementation.

the modifications done during the initial development until a first basic prototype was functional).

Unlike for design testing, we left them a lot of freedom for implementation testing. Hence, we asked how they actually used the generated test suite. 14 out of 19 pairs responded that they simulated the test scenarios, and between each scenario, verified through system/user testing (either manually or not) the general behaviour of the system. The 5 other groups explicitly stated they used unit testing (i.e. simple tests directed at individual classes, features or methods) between the test scenarios.

1. Effectiveness. As before, we evaluate the effectiveness of our testing approach for implementation testing by counting the number of times they modified the implementation of their application in response to the tests. The results are shown in Fig. 9.

We can observe that implementation testing led to less changes to the implementation. Only 9 out of 19 groups found and corrected errors by simulating the test suite, and less than 40% found more than one error. Reasons for this will be discussed later, when discussing weaknesses for implementation testing.

2. Completeness. Theoretically, the pairwise completeness of our testing approach is assured. Hence, we were mostly interested in finding out what were the causes of the errors the participants still detected in their FPCOP application at implementation time. We proposed a variety of possible root causes for the students to choose from. The results shown in Fig. 10 contain the answers only from those 9 pairs who didn't answer "None" to the previous question.

In Fig. 10, "Declaration of the models" refers to the fact that in the RubyCOP implementation language, the designed context–feature model needs to be explicitly declared, which is sometimes the cause of small errors. The "Application classes" are the main classes composing the system, and which the features will refine.

Overall, the causes of errors indicated by the participants are pretty varied, ranging from simple declaration errors to feature errors or feature-interaction problems. Errors were found in

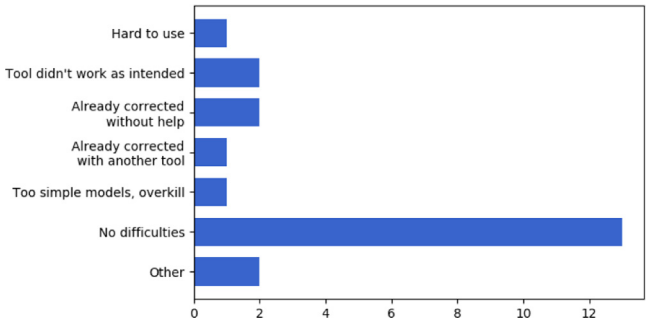

Fig. 11. Difficulties encountered when using the tool or implementation testing.

nearly all parts of the system, apart from the application classes. This is not too surprising as the application classes are used in each and every configuration, and a basic usage of the application was verified before using the generated suite. Two pairs indicated "Other" as source of error but did not provide further information on the error.

3. Weaknesses. In this part of our study we mainly focused on the difficulties encountered when using our tool. The results are shown in Fig. 11.

Overall, there were little or no difficulties in understanding how to use the tool. This can be explained by the fact that they had a lot of liberty in using the tool as they saw fit, and since they already used it for design testing before.

These results, together with the participants' open answers to the questionnaire and the demo of their application, shed more light on why more than half of them didn't find any implementation errors by simulating a generated test suite. They didn't have the time, over two months (but with several other courses in parallel), to design and implement a complex and complete context-aware system. To compensate for the lack of time, some groups limited themselves to prototypes mimicking the actions of complex features, instead of actually implementing them. Moreover, being an experiment of only two months, their system didn't undergo several revisions or changes, thus limiting the appearance of more complex errors.

Due to the lack of depth of the systems implemented by some pairs, simple unit testing often sufficed to discover the majority of errors. E.g., a pair that answered "None" to the number of modifications stated that they "did simple user testing and fixed bugs".

7.2.5. Summary of the experiment

We can conclude that our testing approach is well suited for design testing. It was effective for an overwhelming majority of participant pairs, and helped correcting a variety of errors covering the different models. It gave the participants new ideas and deepened their comprehension of their application. Their feedback also encourages us to further develop and refine our approach into a more complete tool, with better integration and user interface and more supporting resources for beginners.

Regarding implementation testing the results are less conclusive, since the participants focused more on the design phase and had less time to implement a very complex application. Nevertheless the results did confirm that the generated test suites helped to find various types of errors in the actual implementation of FBCOP systems. But a longer and more guided experiment with more focus on the implementation phase is needed to further confirm the effectiveness of our testing approach for implementation testing.

7.3. Summary

We answered RQ3 by demonstrating the use of our testing approach for design testing on our case study. We then validated it further, as well as answered RQ4, through an experiment where 19 pairs of developers used it to prototype an actual FBCOP application. Again, the experiment demonstrated the effectiveness of the approach for design testing. It was less conclusive for implementation testing due to the limited extent of the experiment. The experiment also provided useful input to further improve our tool so that it becomes easier to use in practice.

8. Test suite augmentation

Like any software system, FBCOP systems are subject to continuous evolution that eventually expands the initial system's range of possible behaviours and contexts. This evolution was highlighted in the development methodology proposed in Section 3.4. Those additions, which introduce new contexts, features and relationships in the system's models, call for an augmentation of the test suite. For reasons of efficiency and stability, it is worth trying to achieve this augmentation incrementally, i.e. without recomputing the entire test suite, as was discussed in Section 4.4. The goal of this augmentation is to obtain a *complete* test suite, i.e. which covers all pairs of features in the system (as explained in Section 5), with minimal efforts.

We aim at proposing the least intrusive evolution strategy possible. To this end, inputs are kept to a minimum. The inputs are the previous test suite and the description of the new system (i.e. there is no need to keep track of the changes applied to the system explicitly). In the same spirit, the expected output is simple: a description of what to change or add to the previous test suite. Our answer to **RQ5** is then a greedy algorithm for incremental test suite augmentation that works in two steps:

(1) The algorithm starts by updating existing test scenarios with variables representing new features/contexts that were added. The idea is to try to update each scenario from the original test suite. As each variable can take two values, the addition of n features/contexts generates 2^n configurations from the original scenario, that is one for each possible Boolean assignment. If one such configuration is valid, then we not only keep the pairs from the original test, but we also add pairs that cover the new features/contexts. Note that, since new features/contexts lead to new constraints, this may lead to test scenarios where none of the assignments remain valid. That is, some scenarios from the initial suite may not be expandable and will be dismissed. The output of this step is a list of new context switches and their corresponding feature (de)activations in each of the existing test scenarios, to keep them valid and reuse them.
(2) As second step, the algorithm checks if the augmented test suite is complete (i.e., complete coverage of possible pairs). If not, the algorithm pre-processes the CIT algorithm with the current version of the augmented test suite, i.e. the initial list of test scenarios of the CIT algorithm of Section 5.1 is no longer empty. Finally, the generated test scenarios are then rearranged with the algorithm of Section 6 (updated test scenarios keep their original order). The result is a complete augmented test suite.

8.1. Strategies for step 1

In practice, we do not want to enumerate all 2^n configurations in Step 1. Instead, we propose two strategies to generate a valid configuration for the original test scenarios, if this is possible.

8.1.1. Test update based on SAT solving

We update each scenario from the original suite with Boolean variables associated to the new features/contexts. We can feed this new partially assigned configuration to a SAT solver to try and generate a configuration that satisfies the constraints induced by the new system. The SAT solver will find assignments for the new Boolean variables such that the constraints are satisfied, hence leading to a new valid scenario, if there is one. Non-satisfiable test scenarios are dismissed.

8.1.2. Test update based on features

By default, incremental SAT solving will often assign the new contexts/features to false, or at least generate very similar configurations. That limits the coverage of the new pairs induced by the introduction of new contexts/features, as we do not inject much diversity in the old test suite. We therefore propose to help the SAT solver through the random generation of "partial scenarios". These are partial configurations for which only the variables of the new features/contexts are assigned a value and variables representing features/contexts of the original system are left free. We try to randomise the generation of such partial scenarios so that they uniformly cover the set of new pairs. Such partial configurations are then combined with existing scenarios taken from the original test suite. Of course, due to model constraints, this may lead to invalid configurations. However, as opposed to the problem identified by Cohen et al. (2008), we are not adding constraints after computing the whole array, but just checking compatibility for a restricted number of partial configurations with a fallback strategy (Strategy 1). Hence, we avoid an uncertain and very inefficient regeneration phase.

Assume a test suite containing test scenarios $t_1, \ldots, t_l$, a LIFO queue Q which contains partial scenarios $s_1, \ldots s_m$, and a maximum number of steps S. We present the test expansion procedure below. We iterate over the following four steps until all test scenarios have either been updated or deleted :

(1) Let t_k be the current candidate test scenario to be updated, with $k \in \{1 \ldots l\}$. Pop Q to retrieve partial scenario s_{curr}.
(2) Try to combine s_{curr} with the next S test scenarios $t_k, \ldots, t_{k+S-1}$ consecutively. Stop if one test scenario t_{k+i} is not compatible with partial scenario s_{curr} (checked through SAT solving).
(3) Push s_{curr} to Q.
(4) If scenario t_k has not been updated after m iterations, then this means that we tested all combinations between $s_1, s_2, \ldots, s_m$ and t_k. In such case, apply Strategy 1.

The above algorithm selects t_k, the next test scenario to be updated and a partial scenario s_{curr} (1). It then tries to combine s_{curr} with the next S scenarios (2) in order to identify and update those tests in $t_k \ldots t_{k+S-1}$ that are compatible with this partial scenario. All compatible scenarios are thus considered updated with the values of partial scenario s_{curr}, while the incompatible test scenario (if there is one) is left unchanged. Note that the bigger S is, the less diversity there will be if a lot of scenarios are compatible with s_{curr}. On the other hand, keeping the same partial scenario reduces the number of modifications needed. The algorithm then selects the next partial scenario (3). In case a scenario t_k cannot be updated by any partial scenario, we re-apply Strategy 1 on this test scenario (4).

8.2. Performance

We now evaluate the performance of the above strategies on our case study of Fig. 1. We start with the grey contexts, mapping and features that we consider to be the original system. We then assume the system evolves into a second version that can adapt

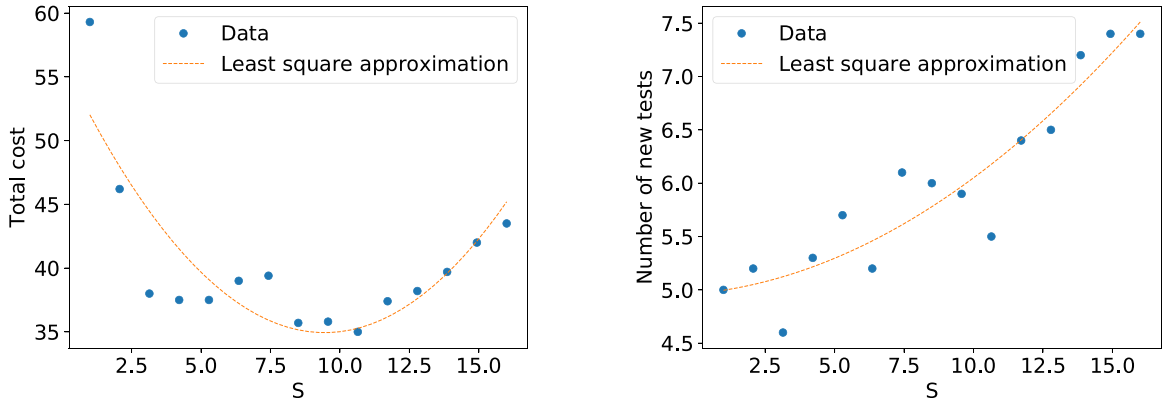

Fig. 12. Influence of parameter S on the total cost of augmenting a test suite.

to a user's age, corresponding to the orange part of Fig. 1. It comprises contexts *Age*, *Teen* and *Adult* and features *AddSystem*, *Match*, *Search*, *ProfilePicture* and *Description*. After that, the system evolves again to include a set of features and contexts to obtain a responsive user interface, corresponding to the red part of Fig. 1. It contains the features *Display*, *Keyboard*, *Layout*, *Minimalist* and *Complete* and the contexts *Device*, *Smartphone*, *Tablet* and *Desktop*.

We define new metrics to study the performance of the augmentation procedure. Assume the original test suite contains l tests. Consider an augmented test suite, containing both updated test scenarios and new generated test scenarios produced by the augmentation procedure. We define the *modification cost* to be the number of new context switches introduced in step (1). We define the *generation cost* to be the number of context switches introduced in step (2) (creation cost of the new test scenarios). The *total cost* is their sum.

We conduct two sets of experiments. In the first one, we explore the impact of the choice of the value of S on Strategy 2. In Fig. 12, we observe the expected compromise: small values of S produce few new test cases (thus low generation cost) but high total cost due to high modification cost. Conversely, high values of S produce many new tests, resulting in high total cost. A compromise can be around the value 9, with a total cost of 35.

In Step 2 of Strategy 2, we combine partial scenarios to a maximum of S consecutive test scenarios to update them while minimising change. We study the actual number of consecutive test scenarios updated with a single partial scenario, on average, or *updates/partial scenario*. In theory, this number should approach S. Instead, in Fig. 13 we observe that this ratio flattens as S increases. Indeed, we have to change the partial scenario used if one incompatibility is found in this Step 2, and the probability of this phenomenon occurring increases with S. Hence, these incompatibilities force the algorithm to use different partial scenarios, no matter how S increases. Observe also that high values of S still tend to decrease the overall diversity of the updated scenarios, as illustrated with the growing number of new test cases needed in Fig. 12.

We now compare both incremental augmentation strategies. Table 13 shows the costs of augmenting the test suite when moving from the original to the second version of the system. Table 14 shows the results for the augmented test suite obtained when moving from the second to the third version. We compare those results with re-applying the CIT algorithm without taking any existing test suites into account (*NOREUSE* row).

NOREUSE produces the least amount of test scenarios, which is expected since CIT's objective is to minimise such number. However, it can be very expensive as the CIT algorithm has to redo the entire work for each incremental evolution. As expected, Strategy 1 injects little diversity in the existing test scenarios (as illustrated by the low modification cost) but adds many more test scenarios. Strategy 2 addresses this problem and achieves the lowest total cost, reducing in the case of Table 14 by more than a factor 4 the cost of creating a new test suite by reusing the previous one.

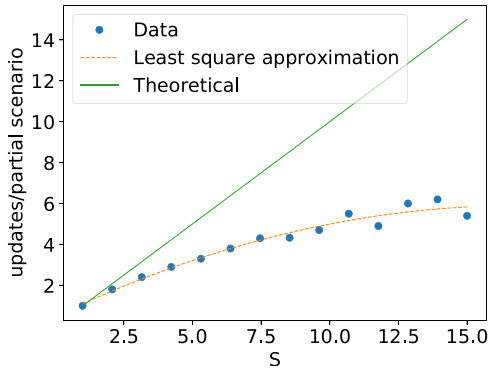

Fig. 13. Number of updates per partial scenarios for each step S, on average.

Table 13
Comparison when going from version 1 to version 2. Best cost and size are highlighted in bold.

Strategy	Modif. cost	Total cost	Size
NOREUSE	/	113	**15.7**
Strategy 1	2	22.5	20.3
Strategy 2	8.33	**11**	15.9

Table 14
Comparison when going from version 2 to version 3. Best cost and size are highlighted in bold.

Strategy	Modif. cost	Total cost	Size
NOREUSE	/	152	**17.4**
Strategy 1	4.9	46	27.4
Strategy 2	5.2	**35**	21.4

8.3. Summary

We answered RQ5 with a greedy algorithm for incremental test suite augmentation, which shows how to update and reuse a previous test suite in order to produce a complete test suite with minimal efforts. We proposed two strategies in order to reduce the number of supplementary test scenarios needed to regenerate the complete pairwise coverage. Finally, we compared them on the creation cost and the number of additional test scenarios needed to fully augment a test suite. By using this solution instead

of generating a completely new test suite, we reduced by more than 75% the cost of creating a test suite upon evolution of the system.

9. Conclusion and future work

9.1. Contributions

We explored the problem of testing feature-based context-oriented programming systems, building upon a pairwise combinatorial interaction testing approach stemming from the domain of software product lines. We answered five research questions: **(RQ1)** How to generate a pertinent yet tractable set of test scenarios for a given FBCOP application? **(RQ2)** How to minimise the effort of simulating these generated test scenarios? **(RQ3)** How efficient is our testing approach to find errors (RQ3a) and what type of errors can we find using it (RQ3b)? **(RQ4)** How relevant and easy-to-use is the testing approach for developers? **(RQ5)** How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?

To answer RQ1 we reduced the problem to one of computing test suites for highly re-configurable systems with constraints and pairwise testing. RQ2 was addressed by rearranging the test scenarios in such a test suite to minimise the number of context switches needed to simulate the entire suite. To answer RQ3 and RQ4, we ourselves used and analysed our testing approach on an actual case study, before conducting an experiment where we analysed how several pairs of developers used it. Finally, RQ5 was answered by proposing an algorithm to incrementally augment existing test suites upon program evolution.

9.2. Threats to validity

Our results are based on an academic case study or were produced in an academic environment. The small academic case study could induce bias, as our testing approach could have performed well on it but be hard to reproduce on different applications. We limited this bias by conducting a larger experiment where 38 students used our approach. However, as they were all from the same academic background, we still need to continue experimenting, in particular with industrial applications which could introduce a new complexity. Fortunately, as the size of all generated test suites grows logarithmically in the size of the number of contexts and features (as explained in Section 5.3), scalability will not be an issue.

RQ5 explores how to augment an already existing test suite. While we propose a first functional solution with promising results, it needs some conditions to be satisfied (only additions of contexts and features, the previous test suite is still valid with the new system) which will not, in practice, always be satisfied. We will need to improve the stability of our solution (i.e. it handles every possible situation optimally and gives feedback) before developing a proper tool and conducting more exhaustive experiments on it.

9.3. Future work

Validation of context-oriented programs is still in its infancy. Many possibilities exist to improve further upon the work presented in this paper. We could improve Cohen's algorithm (Cohen et al., 2008) by exploiting the fact that our SAT formulas are mostly produced from feature diagrams (representing either context or feature models). Exploiting such representation with algorithms such as those presented by Johansen et al. (2012) and Kowal et al. (2016) could help identify dead/core features or other anomalies that should be embedded in or rejected from each test scenario. Another alternative would be to use Quantified Boolean Formula solvers (Mauro, 2021). Recent SAT solvers could compute randomly valid configurations of SAT formulas and have been shown to be efficient to estimate the t-wise coverage of various large size examples (Plazar et al., 2019; Baranov et al., 2020) but have not yet been adapted to handle systems with constraints.

We could also take inspiration from alternative approaches towards pairwise testing for software product lines (Perrouin et al., 2012) and more diverse testing strategies (do Carmo Machado et al., 2012). Other closely related fields worth exploring further are multi-objective (Henard et al., 2013) and many-objective test generation (Hierons et al., 2020). More distant software testing approaches like mutation testing (Reuling et al., 2015) and hybrid concolic testing (Souto et al., 2017) could be adapted to or serve as inspiration for testing FBCOP systems.

Whereas the focus of this paper was on test scenario generation for feature-based context-oriented programs, we believe the approach could be generalised to other context-aware and self-adaptive systems if we would manage to model them in terms of a discretised set of contexts and features. Kotsiantis et al. provide a broad survey of possible discretisation techniques (Kotsiantis and Kanellopoulos, 2006). On a longer term, it would also be worth exploring systems whose contexts and feature activation depends on quantitative information. This includes, e.g., the new line of research dedicated to real-time and probabilistic product lines (Göttmann et al., 2020; Dubslaff et al., 2014; Cordy et al., 2012, 2021).

We proposed various simple metrics to compare the effectiveness of our testing approach. More complex metrics could be devised that provide a more accurate assessment of different aspects of the quality of the produced test scenarios (efficiency, pertinence, testability, various forms of coverage in terms of contexts and features and combinations thereof), not only to compare a rearranged test suite with the original one, but also to get an idea of how good a test suite is for testing a given system, or how the quality of the tests evolve as the system evolves.

The rearrangement algorithm itself could also be extended by considering other objectives. In the spirit of Devroey et al. (2017), one could target rearrangements that prioritise contexts that have been identified to be used frequently in deployed systems. We could also explore strategies that give more importance to some pre-defined high-frequency switches between contexts, or to the cost of (un)deploying features during the program's execution. Similarly, new ways to generate partial scenarios in the augmentation procedure could be considered.

The control flow of a feature-based context-oriented system 3.3 could be improved by developing a procedure that uses our new results in using a SAT solver. It would allow us to fuse the constraints while keeping intact the advantages of a clear separation of context and feature.

The main objective of this paper was to pose the foundations of a testing process dedicated to feature-based context-oriented applications. So far, our testing approach has been applied to smaller academic case studies only, yet we hope to study its scalability to industrial-size applications in the future. During our experiment, our testing approach proved useful mainly for design testing, and to a lesser extent for implementation testing purposes, and its results are promising. The cost of creating a test suite for our case study has been reduced by more than half as compared to no reordering. The cost of creating a test suite has been reduced by 75% using test suite augmentation, as compared to no reuse of a previous test suite.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A public repository (github) is linked in the paper and it contains the data used in the experiment.

References

Abowd, Gregory D., Dey, Anind K., Brown, Peter J., Davies, Nigel, Smith, Mark, Steggles, Pete, 1999. Towards a better understanding of context and context-awareness. In: Gellersen, Hans-W. (Ed.), Handheld and Ubiquitous Computing. Springer, Berlin, Heidelberg, pp. 304–307.

Acher, Mathieu, Collet, Philippe, Fleurey, Franck, Lahire, Philippe, Moisan, Sabine, Rigault, Jean-Paul, 2009. Modeling context and dynamic adaptations with feature models. In: 4th International Workshop Models@Run.Time at Models 2009. MRT'09, United States, p. 10, URL https://hal.archives-ouvertes.fr/hal-00419990.

Al-Hajjaji, Mustafa, Thüm, Thomas, Lochau, Malte, Meinicke, Jens, Saake, Gunter, 2019. Effective product-line testing using similarity-based product prioritization. Software Syst. Model. 18 (1), 499–521.

Bagheri, Ebrahim, Gasevic, Dragan, 2011. Assessing the maintainability of software product line feature models using structural metrics. Softw. Qual. J. 19 (3), 579–612.

Baranov, Eduard, Legay, Axel, Meel, Kuldeep S., 2020. Baital: An adaptive weighted sampling approach for improved t-wise coverage. In: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE '20, ACM, pp. 1114–1126.

Baresi, Luciano, Quinton, Clément, 2015. Dynamically evolving the structural variability of dynamic software product lines. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 57–63.

Benavides, David, Segura, Sergio, Ruiz-Cortés, Antonio, 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636.

Bencomo, Nelly, Lee, Jaejoon, Hallsteinsen, Svein, 2010. How dynamic is your dynamic software product line?

Bencomo, Nelly, Sawyer, Peter, Blair, Gordon S., Grace, Paul, 2008. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: SPLC (2). pp. 23–32.

Bryce, Renée C, Sampath, Sreedevi, Pedersen, Jan B, Manchester, Schuyler, 2011. Test suite prioritization by cost-based combinatorial interaction coverage. Int. J. Syst. Assur. Eng. Manag. 2 (2), 126–134.

Capilla, Rafael, Bosch, Jan, Trinidad, Pablo, Ruiz-Cortés, Antonio, Hinchey, Mike, 2014a. An overview of dynamic software product line architectures and techniques: Observations from research and industry. J. Syst. Softw. 91, 3–23.

Capilla, Rafael, Hinchey, Mike, Díaz, Francisco J., 2015. Collaborative context features for critical systems. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems. pp. 43–50.

Capilla, Rafael, Ortiz, Óscar, Hinchey, Mike, 2014b. Context variability for context-aware systems. Computer 47 (2), 85–87.

Cardozo, Nicolás, Günther, Sebastian, DHondt, Theo, Mens, Kim, 2011. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In: International Conference on Software Engineering Advances. ICSEA'11, IARIA, pp. 130–135.

Cardozo, Nicolás, Mens, Kim, Orban, Pierre-Yves, González, Sebastián, De Meuter, Wolfgang, 2014. Features on demand. In: Proceedings of 8th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '14, ACM, pp. 18:1–18:8.

Catal, Cagatay, Mishra, Deepti, 2013. Test case prioritization: A systematic mapping study. Softw. Qual. J. 21 (3), 445–478.

Cohen, David M., Dalal, Siddhartha R., Fredman, Michael L., Patton, Gardner C., 1997. The AETG system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23 (7), 437–444.

Cohen, Myra B., Dwyer, Matthew B., Shi, Jiangfan, 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. Softw. Eng. 34 (5), 633–650.

Cordy, Maxime, Classen, Andreas, Heymans, Patrick, Legay, Axel, Schobbens, Pierre-Yves, 2013a. Model checking adaptive software with featured transition systems. In: Cámara, Javier, de Lemos, Rogério, Ghezzi, Carlo, Lopes, Antónia (Eds.), Assurances for Self-Adaptive Systems - Principles, Models, and Techniques. In: Lecture Notes in Computer Science, vol. 7740, Springer, pp. 1–29. http://dx.doi.org/10.1007/978-3-642-36249-1_1.

Cordy, Maxime, Lazreg, Sami, Papadakis, Mike, Legay, Axel, 2021. Statistical model checking for variability-intensive systems: Applications to bug detection and minimization. Formal Aspects Comput. 33 (6), 1147–1172. http://dx.doi.org/10.1007/s00165-021-00563-2.

Cordy, Maxime, Legay, Axel, Schobbens, Pierre-Yves, Traonouez, Louis-Marie, 2013b. A framework for the rigorous design of highly adaptive timed systems. In: 1st FME Workshop on Formal Methods in Software Engineering. FormaliSE 2013, San Francisco, CA, USA, May 25, 2013, IEEE Computer Society, pp. 64–70. http://dx.doi.org/10.1109/FormaliSE.2013.6612279.

Cordy, Maxime, Schobbens, Pierre-Yves, Heymans, Patrick, Legay, Axel, 2012. Behavioural modelling and verification of real-time software product lines. In: de Almeida, Eduardo Santana, Schwanninger, Christa, Benavides, David (Eds.), 16th International Software Product Line Conference, Vol. 1. SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1, ACM, pp. 66–75. http://dx.doi.org/10.1145/2362536.2362549.

Costanza, Pascal, D´Hondt, Theo, 2008. Feature descriptions for context-oriented programming. In: Lero Int. Science Centre. pp. 9–14.

Costanza, Pascal, Hirschfeld, Robert, 2005. Language constructs for context-oriented programming: An overview of contextL. In: Proceedings of the 2005 Symposium on Dynamic Languages. DLS '05, ACM, pp. 1–10.

Coutaz, Joëlle, Crowley, James L., Dobson, Simon, Garlan, David, 2005. Context is key. Commun. ACM 48 (3), 49–53.

Czarnecki, Krzysztof, Wasowski, Andrzej, 2007. Feature diagrams and logics: There and back again. In: 11th International Software Product Line Conference. SPLC '07, IEEE, pp. 23–34.

Desmet, Brecht, Vallejos, Jorge, Costanza, Pascal, De Meuter, Wolfgang, D'Hondt, Theo, 2007. Context-oriented domain analysis. In: Modeling and using Context. Springer, pp. 178–191.

Devroey, Xavier, Perrouin, Gilles, Cordy, Maxime, Samih, Hamza, Legay, Axel, Schobbens, Pierre-Yves, Heymans, Patrick, 2017. Statistical prioritization for software product line testing: An experience report. Software Syst. Model. 16 (1), 153–171.

Dinkelaker, Tom, Mitschke, Ralf, Fetzer, Karin, Mezini, Mira, 2010. A dynamic software product line approach using aspect models at runtime. In: 5th Domain-Specific Aspect Languages Workshop.

do Carmo Machado, Ivan, McGregor, John D, Santana de Almeida, Eduardo, 2012. Strategies for testing products in software product lines. ACM SIGSOFT Software Eng. Not. 37 (6), 1–8.

Dubslaff, Clemens, Klüppelholz, Sascha, Baier, Christel, 2014. Probabilistic model checking for energy analysis in software product lines. In: Binder, Walter, Ernst, Erik, Peternier, Achille, Hirschfeld, Robert (Eds.), 13th International Conference on Modularity. MODULARITY '14, Lugano, Switzerland, April 22-26, 2014, ACM, pp. 169–180. http://dx.doi.org/10.1145/2577080.2577095.

Duhoux, Benoît, 2022. Feature-Based Context-Oriented Software Development (Ph.D. thesis). Université catholique de Louvain.

Duhoux, Benoît, Mens, Kim, Dumas, Bruno, 2018. Feature visualiser: An inspection tool for context-oriented programmers. In: Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition. COP '18, Amsterdam, Netherlands, Association for Computing Machinery, New York, NY, USA, pp. 15–22.

Duhoux, Benoît, Mens, Kim, Dumas, Bruno, 2019a. Implementation of a feature-based context-oriented programming language. In: Proceedings of the Workshop on Context-Oriented Programming. COP '19, ACM, pp. 9–16.

Duhoux, Benoît, Mens, Kim, Dumas, Bruno, Leung, Hoo Sing, 2019b. A context and feature visualisation tool for a feature-based context-oriented programming language. In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution. SATTOSE '19, In: CEUR Workshop Proceedings, vol. 2510, CEUR-WS.org.

Fernandes, Paula, Werner, Cláudia, Teixeira, Eldânae, 2011. An approach for feature modeling of context-aware software product line. J. Univers. Comput. Sci. 17 (5), 807–829.

González, Sebastián, Cardozo, Nicolás, Mens, Kim, Cádiz, Alfredo, Libbrecht, Jean-Christophe, Goffaux, Julien, 2011. Subjective-C: Bringing context to mobile platform programming. In: Proceedings of 3rd International Conference on Software Language Engineering. SLE '10, Springer, pp. 246–265.

Göttmann, Hendrik, Luthmann, Lars, Lochau, Malte, Schürr, Andy, 2020. Real-time-aware reconfiguration decisions for dynamic software product lines. In: Lopez-Herrejon, Roberto Erick (Ed.), SPLC '20: 24th ACM International Systems and Software Product Line Conference, Vol. A. Montreal, Quebec, Canada, October 19-23, 2020, ACM, pp. 13:1–13:11. http://dx.doi.org/10.1145/3382025.3414945.

Hartmann, Herman, Trew, Tim, 2008. Using feature diagrams with context variability to model multiple product lines for software supply chains. In: Proceedings of 12th International Software Product Line Conference. SPLC '08, IEEE, pp. 12–21.

Henard, Christopher, Papadakis, Mike, Perrouin, Gilles, Klein, Jacques, Traon, Yves Le, 2013. Multi-objective test generation for software product lines. In: Proceedings of the 17th International Software Product Line Conference. pp. 62–71.

Hierons, Robert M, Li, Miqing, Liu, Xiaohui, Parejo, Jose Antonio, Segura, Sergio, Yao, Xin, 2020. Many-objective test suite generation for software product lines. ACM Trans. Software Eng. Methodol. (TOSEM) 29 (1), 1–46.

Hirschfeld, Robert, Costanza, Pascal, Haupt, Michael, 2008a. Generative and Transformational Techniques in Software Engineering II. Springer, pp. 396–407, Chapter An Introduction to Context-Oriented Programming with ContextS.

Hirschfeld, Robert, Costanza, Pascal, Nierstrasz, Oscar, 2008b. Context-oriented programming. J. Object Technol. 7 (3), 125–151.

Johansen, Martin Fagereng, Haugen, Øystein, Fleurey, Franck, 2012. An algorithm for generating T-wise covering arrays from large feature models. In: Proceedings of the 16th International Software Product Line Conference - Volume 1. SPLC '12, ACM, pp. 46–55.

Kang, Kyo C., Cohen, Sholom G., Hess, James A., Novak, William E., Peterson, A. Spencer, 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, Carnegie-Mellon University Software Engineering Institute.

Kästner, Christian, Apel, Sven, 2011. Feature-oriented software development. In: International Summer School on Generative and Transformational Techniques in Software Engineering. Springer, pp. 346–382.

Kotsiantis, Sotiris B., Kanellopoulos, Dimitris N., 2006. Discretization techniques: A recent survey.

Kowal, Matthias, Ananieva, Sofia, Thüm, Thomas, 2016. Explaining anomalies in feature models. ACM SIGPLAN Not. 52 (3), 132–143.

Lee, Jihyun, Kang, Sungwon, Lee, Danhyung, 2012. A survey on software product line testing. In: Proceedings of the 16th International Software Product Line Conference-Volume 1. pp. 31–40.

Lity, Sascha, Al-Hajjaji, Mustafa, Thüm, Thomas, Schaefer, Ina, 2017. Optimizing product orders using graph algorithms for improving incremental product-line analysis. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems. pp. 60–67.

Marques-Silva, Joao P., Sakallah, Karem A., 1999. GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comput. 48 (5), 506–521.

Martou, Pierre, Mens, Kim, Duhoux, Benoît, Legay, Axel, 2021. Towards a testing approach for feature-based context-oriented systems. In: Proceedings of the Thirteenth International Conference on Advances in System Testing and Validation Lifecycle. VALID'21, IARIA, pp. 1–11.

Mauro, Jacopo, 2021. Anomaly detection in context-aware feature models. In: 15th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS '21, pp. 1–9.

Mauro, Jacopo, Nieke, Michael, Seidl, Christoph, Chieh Yu, Ingrid, 2018. Context-aware reconfiguration in evolving software product lines. Sci. Comput. Program. 163, 139–159.

Mendonca, Marcilio, Branco, Moises, Cowan, Donald, 2009. SPLOT: Software product lines online tools. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. pp. 761–762.

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

Mens, Kim, Cardozo, Nicolás, Duhoux, Benoît, 2016. A context-oriented software architecture. In: Proceedings of the 8th International Workshop on Context-Oriented Programming. COP '16, Rome, Italy, ACM, New York, NY, USA, pp. 7–12.

Morin, Brice, Barais, Olivier, Nain, Gregory, Jézéquel, Jean-Marc, 2009. Taming dynamically adaptive systems using models and aspects. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 122–132.

Perrouin, Gilles, Oster, Sebastian, Sen, Sagar, Klein, Jacques, Baudry, Benoit, Le Traon, Yves, 2012. Pairwise testing for software product lines: Comparison of two approaches. Softw. Qual. J. 20 (3), 605–643.

Plazar, Quentin, Acher, Mathieu, Perrouin, Gilles, Devroey, Xavier, Cordy, Maxime, 2019. Uniform sampling of SAT solutions for configurable systems: Are we there yet? In: 12th IEEE Conference on Software Testing, Validation and Verification. ICST '19, IEEE, pp. 240–251.

Reuling, Dennis, Bürdek, Johannes, Rotärmel, Serge, Lochau, Malte, Kelter, Udo, 2015. Fault-based product-line testing: Effective sample generation based on feature-diagram mutation. In: Proceedings of the 19th International Conference on Software Product Line. SPLC '15, Nashville, Tennessee, Association for Computing Machinery, New York, NY, USA, pp. 131–140. http://dx.doi.org/10.1145/2791060.2791074.

Rosenmüller, Marko, Siegmund, Norbert, Pukall, Mario, Apel, Sven, 2011. Tailoring dynamic software product lines. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering. GPCE '11, Portland, Oregon, USA, Association for Computing Machinery, New York, NY, USA, pp. 3–12.

Salvaneschi, Guido, Ghezzi, Carlo, Pradella, Matteo, 2011. JavaCtx: Seamless toolchain integration for context-oriented programming. In: Proceedings of 3rd International Workshop on Context-Oriented Programming. COP '11, ACM, pp. 4:1–4:6.

Salvaneschi, Guido, Ghezzi, Carlo, Pradella, Matteo, 2012. Context-oriented programming: A software engineering perspective. J. Syst. Softw. 85 (8), 1801–1817.

Sánchez, Ana B., Segura, Sergio, Ruiz-Cortés, Antonio, 2014. A comparison of test case prioritization criteria for software product lines. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE, pp. 41–50.

Souto, Sabrina, d'Amorim, Marcelo, Gheyi, Rohit, 2017. Balancing soundness and efficiency for practical testing of configurable systems. In: Proceedings of the 39th International Conference on Software Engineering. ICSE '17, Buenos Aires, Argentina, IEEE Press, pp. 632–642. http://dx.doi.org/10.1109/ICSE.2017.64.

Srikanth, Hema, Cohen, Myra B., Qu, Xiao, 2009. Reducing field failures in system configurable software: Cost-based prioritization. In: 2009 20th International Symposium on Software Reliability Engineering. IEEE, pp. 61–70.

ter Beek, Maurice H., Legay, Axel, 2019. Quantitative variability modeling and analysis. In: Weyns, Danny, Perrouin, Gilles (Eds.), Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems. VAMOS 2019, Leuven, Belgium, February 06-08, 2019, ACM, pp. 13:1–13:2. http://dx.doi.org/10.1145/3302333.3302349.

Ter Beek, Maurice H., Legay, Axel, Lafuente, Alberto Lluch, Vandin, Andrea, 2020. A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Trans. Softw. Eng. 46 (3), 321–345.

Thevenin, David, Coutaz, Joëlle, 1999. Plasticity of user interfaces: Framework and research agenda. In: Interact, vol. 99, pp. 110–117.