Test Scenario Generation for Context-Oriented Programs

Pierre Martou
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
pierre.martou@uclouvain.be

Kim Mens
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Benoît Duhoux
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
benoit.duhoux@uclouvain.be

Axel Legay
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

***Abstract*—Their highly adaptive nature and the combinatorial explosion of possible configurations makes testing context-oriented programs hard. We propose a methodology to automate the generation of test scenarios for developers of feature-based context-oriented programs. By using combinatorial interaction testing we generate a covering array from which a small but representative set of test scenarios can be inferred. By taking advantage of the explicit separation of contexts and features in such context-oriented programs, we can further rearrange the generated test scenarios to minimise the reconfiguration cost between subsequent scenarios. Finally, we explore how a previously generated test suite can be adapted incrementally when the system evolves to a new version. By validating these algorithms on a small use case, our initial results show that the proposed test generation approach is efficient and beneficial to developers to test and improve the design of context-oriented programs.**

***Index Terms*—context-oriented programming, combinatorial interaction testing, covering array, test scenario generation, satisfiability (SAT) checking**

I. Introduction

Context-Oriented Programming (COP) languages help developers build context-aware applications that, based on contextual information sensed from the surrounding environment, can adapt their behaviour dynamically. A wide range of COP languages have been proposed to ease the design and implementation of such applications [1]–[4]. *Feature-Based Context-Oriented Programming* (FBCOP) [5], [6] is a particular class of such languages that takes inspiration from context-oriented programming [7]–[10], Feature Modelling (FM) [11] and Dynamic Software Product Lines (DSPL) [12]–[14].

Whereas COP and FBCOP languages claim to provide better support for dynamic adaptability 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 context in which the application operates.

This paper pursues the following three research questions, each of which coincides with one of the contributions of this paper.

RQ1 How to generate a pertinent yet tractable set of test scenarios for a given FBCOP application?

RQ2 How to minimise the effort of creating these generated test scenarios?

RQ3 How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?

To address RQ1 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 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 [15].

Whereas the algorithm explored to address 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 desire the scenarios to be automatically ordered in such a way that 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 minimize this cost, effectively reducing the effort for a software tester to implement these tests.

Finally, regarding RQ3, we want to avoid having to regenerate completely the test scenarios when new contexts or features appear as the program evolves to a new version. Our approach consists of updating test scenarios that were already generated for the previous version of the evolved program. Hoping that it already contains many of the new pairs that should be covered, Cohen's algorithm [15] is then fed with this candidate test suite and produces 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 each of these three contributions and applied them on a small exploratory case study. Our initial experiments confirm both the feasibility and efficiency of our approach, effectively addressing each of our three research questions. The three contributions combined provide a lightweight methodology and set of algorithms for creating test suites easily and efficiently while developing (feature-based) context-oriented programs.

The remainder of this paper is structured as follows. Section II introduces the FBCOP approach on top of which we built our test scenario generation approach, as well as the case study used as running example and validation. Section III revisits our three research questions and establishes our goals. Section IV through VI then explain and validate each of our three contributions in detail. Section VII concludes the paper and presents interesting avenues of future work.

II. Feature-based context-oriented programming

Unlike other COP languages, FBCOP makes an explicit distinction between how contexts (reifying particular situations sensed from the surrounding environment) and features (behavioural variations specific to certain contexts) are modelled and handled [5], [6].

Before introducing FBCOP, we introduce the running example used throughout this paper and the notion of feature diagrams. Then we exemplify how FBCOP can be used to write programs that adapt their behaviour upon context changes.

A. Smart messaging system

As a running example of a FBCOP program, we implemented a prototype of a *smart messaging system*. The system allows users to exchange messages and is smart in the sense that it can adapt or refine its behaviour depending on some contextual situations. The system's main functionality consists of *Sending* or *Receiving* messages to and from other users. By default, we assume the messages are of type *Text* though richer messages (*Vocal* or *Photo*) can be exchanged depending on the status of the internet *Connection* or the *Peripheral* cards (audio or video) installed on the *Device* running the program.

Conversations can take place between a *Group* of *Friend*s. Each user is identified by its unique *FriendName*. Depending on the users' *Age* (*Teen* or *Adult*), they can complete their user profile with a textual *Description* or *ProfilePicture* (respectively). How new friends can be added to a conversation also depends on the users' *Age*.

Whenever users receive a new message, a *Notification* may be given via a sound *Alarm* or their device's *Vibration* mode. What notification mode is used depends on the type of *Device*, the *User Availability* and the ambient *Noise* level. The *Device* type also affects other features of the smart messaging system, such as adapting the layout that displays the information or adding a virtual keyboard.

B. Feature diagram

Taking inspiration from Hartmann and Trew's *multiple product line feature model* [12], in feature-based context-oriented programming, both the contexts and features of a FBCOP program are designed as feature diagrams. A feature diagram [11] describes the commonalities and variabilities of a system in terms of a tree-like structure where nodes represent features (or contexts) and edges the constraints between them. Examples of feature diagrams representing, respectively, the context and feature model of our smart messaging system, can be found on the left and right hand side of Figure 1.

Constraints can either be hierarchical or cross-tree constraints. One kind of hierarchical constraints are *mandatory* constraints ensuring that a given child feature is always present if its parent feature is selected. Examples of such features in Figure 1 are *Sending* or *Receiving*. *Optional* constraints express that a child may be selected if its parent is, as for the *Photo* or *Vocal* features. *Or* (resp. *alternative*) constraints impose that at least (resp. exactly) one child should be selected if the parent is. An example of an *or* constraint is the *Display* feature: the messaging system can either display a *Layout*, a *Keyboard* or both to its users. The kind of *Layout* is an example of an *alternative* constraint since the system will use either a *Complete* or a *Minimalist* layout but never both simultaneously.

Cross-tree constraints can be used to express exclusions or requirements between features. An *exclusion* constraint between features ensures only one feature is selected in the system at the same time. A *requirement* constraint between two features means the source feature can be selected only if the target feature is already selected.

A configuration of a feature diagram is a selection of features and is said to be valid if the selection satisfies all the constraints imposed by the diagram.

C. FBCOP architecture

Figure 2 sketches the overall FBCOP architecture. Using our running example we will explain how this architecture enables a FBCOP system to adapt its behaviour to sensed context changes. For more information on how the FBCOP architecture is designed and implemented, please consult our paper [5].

When a particular situation in the surrounding environment is (no longer) sensed, the system attempts to (de)activate it in its context model. This happens during the *context activation* phase. For that, the system updates the current configuration of the context model, representing the current state of the surrounding environment, with the newly (de)activated context. If the updated configuration still satisfies all constraints of the context model, this new configuration is kept and the system proceeds to the second *feature selection* phase. Otherwise, the system rolls back to the previous valid configuration of the context model and ignores the new context. Based on our case study, assume an *Adult* uses the smart messaging system on her *Smartphone*, is currently *Available* for conversation, and that the ambient *Noise* level is *Normal*. As this configuration is valid, the system activates these contexts.

Next, the system determines what features are triggered by these new contexts through the mapping model. This mapping model consists of a conjunction of individual mappings, each

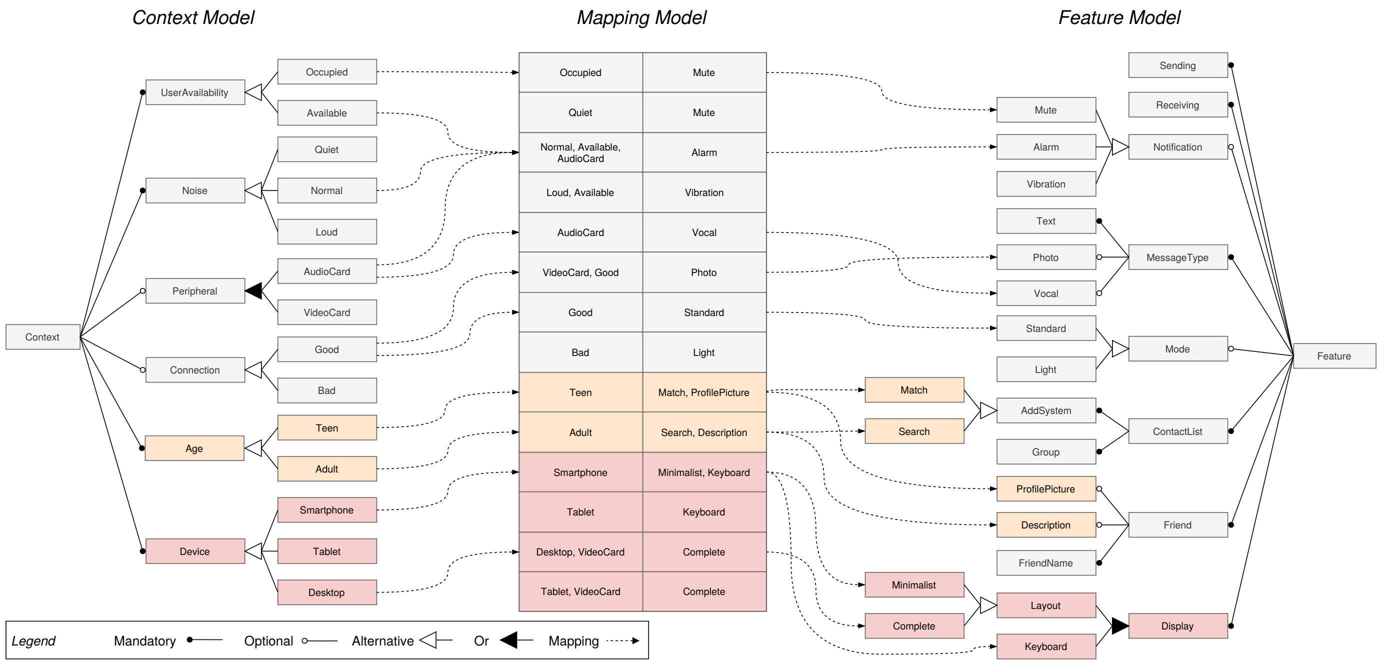

Fig. 1. Design of a prototype of our smart messaging system. The context model is on the left, the feature model on the right and the mapping model between them in the middle. We voluntarily omitted some mapping details for readability reasons.

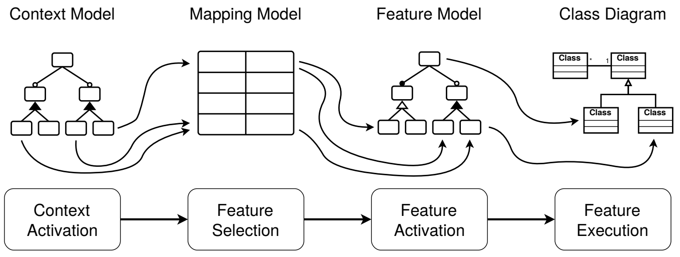

Fig. 2. Overview of the different models and phases in a feature-based context-oriented system [5].

representing a *N:N* relation from contexts to features. In our example, the system would select the following features: *Search*, *Description*, *Minimalist*, *Keyboard*. This selection then gets processed by the system in the *feature activation* phase to attempt to activate (or deactivate) the corresponding features in its feature model. Similar to the *context activation* phase, the system updates the current configuration of the feature model, describing the currently active features, with the newly (de)activated features. If the updated configuration is valid, the system keeps this updated configuration. Otherwise it ignores this attempt. In our example, as the new configuration of the feature model remains valid, the system activates this selection.

Finally, during the *feature execution* phase, the system adapts its behaviour by installing (resp. removing) the code associated to the selected features in the running code. In our example, after installing these features, the user will be able to *Search* for friends to add them to a group conversation, as well as to add a detailed *Description* to her user profile. As she uses a *Smartphone*, a *Minimalist* layout and virtual *Keyboard* will be presented by the system.

III. Testing

Having introduced the notions of context-oriented programming and feature-based context-oriented programming, and illustrated them on our running example, let us revisit our three RQs in detail.

A. Test suite generation

Because of the large number of possible combinations of contexts and features, and the dynamicity of the contexts that trigger these features, exhaustively testing all possible scenarios induced by a such a program becomes intractable. The 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 features to activate or deactivate. It is worth noting that in the feature-based context-oriented approach considered, contexts (and features) are binary entities 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 child contexts such as ***Quiet***, ***Normal*** and ***Loud*** in an alternative relationship. For simplicity's sake, let us assume that we have an application with these three contexts ***Quiet***, ***Normal*** and ***Loud*** and two features *Photo* and *Alarm*.

TABLE I
EXAMPLE OF A TEST SUITE, WHERE *A* (RESP. *D*) MEANS THAT A PARTICULAR CONTEXT OR FEATURE IS *A*CTIVATED (RESP. *D*EACTIVATED).

Scenario	**Quiet**	**Normal**	**Loud**	Alarm	Photo
1	D	A	D	A	A
2	A	D	D	A	D
3	D	A	A	D	D
...	...				

Table I lists a subset of the 2^5 possible combinations of these contexts and features. For clarity, we keep the contexts on the left, the features on the right and highlight the contexts in bold. Some of these scenarios may be invalid when they do not respect the constraints imposed by the context, feature or mapping model. Given the *alternative* constraint in the context model between the contexts ***Quiet***, ***Normal*** and ***Loud***, only one of them can be active at a time, making the third scenario invalid. The second scenario is invalid since the mapping does not permit the feature *Alarm* to be active in context ***Quiet***.

In Section IV, we will address RQ1 by reducing an FBCOP system to a highly reconfigurable system in presence of constraints, in order to follow a pairwise testing approach [15]. Such computation exploits a SAT-based representation unifying the context, mapping and feature models of an 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 IV-B on complexity, and Section IV-C which will highlight the high efficiency of two well-known optimizations on FBCOP systems.

To our knowledge, there is little research on this subject and focused on context-oriented programming. Whereas in this paper we mainly rely on Cohen's approach [15], in the future we will further improve our approach by taking insights from pairwise testing in software product lines [16] and more diverse testing approaches [17]. Close fields include for example multi-objective [18] or many-objective test generation [19].

B. Creation cost of a suite

The presentation of scenarios as shown in Table I quickly becomes 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 of contexts and features between subsequent scenarios instead. This idea is illustrated in Table II.

TABLE II
EXAMPLE OF A TEST SUITE WITH ACTIVATION SWITCHES

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

In addition to being more compact, this representation is closer to how an FBCOP system works 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 thus of fundamental importance if one wants to reduce simulation costs, which leads us to the second research question (RQ2): **How to minimise the effort of creating these generated test scenarios?**

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 above example has 6 switches. By swapping scenario 1 and 2, the creation cost would get reduced to 4, as illustrated in Table III. In Section V we will generalise this observation into a lightweight test suite rearrangement algorithm, based on the creation cost, a metric tailored to FBCOP. Other rearrangements were studied previously in software product lines, such as using statistical prioritization [20], graph representation and heuristics minimising the analysis effort [21], or similarity-based product prioritisation improving feature interaction coverage as fast as possible during testing [22].

TABLE III
REARRANGED VERSION OF THE EXAMPLE TEST SUITE

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

C. System evolution

Finally, it is inevitable that developers will continue to evolve their system. 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 combinations of interest for the initial system design, this evolution could be done without recomputing the entire suite. This prompts the third research question (RQ3): **How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?**

Let us illustrate this situation by augmenting the example of Table I. 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[1] by keeping the same scenarios and adding a column ***Tablet*** and *Complete* and then assigning them some activation choices (in italics) in Table IV.

Unfortunately, with these values, the first scenario is no longer valid. Indeed, if feature *Photo* is active, then according to the mapping, context ***VideoCard*** must be active too. Moreover, if both ***VideoCard*** and ***Tablet*** are active, then feature *Complete* must be active too, which is inconsistent with its assigned activation choice *D*.

[1]Due to space limitations we replaced the original contexts ***Quiet*** and ***Normal*** and their assignments by "..."

TABLE IV
EXAMPLE OF AN AUGMENTED TEST SUITE

Scenario	...	**Loud**	***Tablet***	Alarm	Photo	*Complete*
1	...	D	*A*	A	A	*D*
2	...	D	*A*	A	D	*A*
3	...	A	*D*	D	D	*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 VI will present such an augmentation algorithm including two different update strategies.

D. Overview

When programming FBCOP systems, developers need system testing in order to debug and evolve their system. The goal of our proposed testing methodology is to help them with the definition of interesting and evolving test suites.

RQ1 will be addressed in Section IV. Our algorithms for rearranging a test suite (RQ2) and incrementally augmenting it (RQ3) are the subject of Sections V and VI, respectively. Together, the proposed algorithms form a complete process schematized in Figure 3.

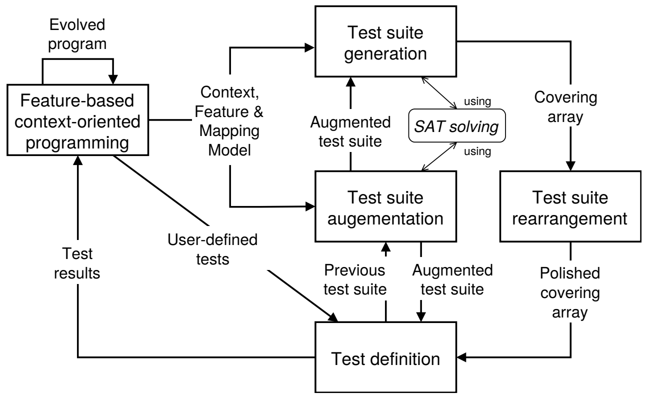

Fig. 3. Summary of our FBCOP testing methodology

IV. 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 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 [15]. This 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 A 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.[2] 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 $f_2,\dots,f_k$ to each remaining feature.
3) Assume all features $f_1,\dots,f_{n-1}$ have been assigned to $v_1, \dots, v_{n-1}$ 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 $((f_1,v_1), (f_n,v_n)), \dots, ((f_{n-1},v_{n-1}), (f_n,v_n))$ are still uncovered and that adding v_n to the partial assignment keeps it valid (checked through SAT solving).

A. Covering Array Generation for FBCOP

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 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 II-A and depicted in Figure 1, a FBCOP system consists of a context model, mapping model and feature model. Both the context and feature models are represented as *feature diagrams* (FD) representing a set of combinations of activated or deactivated (contexts or) features. In this representation, (contexts or) features can be modeled as Boolean variables whose assignments represent their activation status. E.g., if a Boolean variable representing the feature *MessageType* is set to 1 (resp. 0), this means that this feature is activated (resp. deactivated). Any FD can be turned into conjunctive normal form [23] whose Boolean atoms are the Boolean variables corresponding to the features' activation status. Any valid assignment of such a set of constraints represents the list of (de)activated (contexts or) features from the FD.

Now that we know that the feature diagrams for both the context and feature model can be represented by constraints, the only thing left to do is to convert the mapping model into a set of constraints as well. The set of constraints will then be the conjunction of all constraints generated by each of the three models of a FBCOP system, and the list of contexts or

[2]M is an input parameter to Cohen's algorithm which we experimentally assigned to 30 for our case study.

features to activate or deactivate for a given test scenario will be modeled by Boolean variables.

TABLE V
SUBSET OF THE MAPPING MODEL OF OUR CASE STUDY

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

Converting our mapping model to a set of constraints is straightforward. Remember that the mapping can be represented as set of N:N correspondences between contexts and features, as exemplified in Table V. A naive way to convert the mapping represented by this table to propositional logic would be to take each line as a simple equivalence relation between the left and right columns, but this conversion is actually incorrect. Indeed, the equivalence between *Occupied* and *Mute* and the one between *Quiet* and *Mute* would imply the equivalence of the two contexts *Occupied* and *Quiet*. As contexts *Quiet* and *Loud* are mutually exclusive (they are part of an alternative constraint), context *Occupied* would 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. We therefore propose a more liberal conversion where the activated contexts imply the activation of the corresponding features:

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

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

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

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

In order to guarantee a correct mapping, one also needs to make sure 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 :

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

$$Alarm \Rightarrow Normal \wedge AudioCard \wedge Available \tag{6}$$

$$Vibration \Rightarrow Loud \wedge Available \tag{7}$$

From these two sets of Boolean formulas in propositional logic, we can obtain a set of CNF formulas, and thereby complete our conversion 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. In Table VI, we show three test scenarios of a generated test suite for the example presented in Section II-A. The whole test suite has 18 scenarios, and its creation cost (i.e., the number of context switches) is 161. Over 30 different runs of the CIT algorithm, the average size and cost of the generated scenarios are 17.36 and 151.9, respectively.

Interestingly, Scenario 13 in Table VI, while correct, highlights an error in the design of our case study: it seems possible to have a *Tablet* without an appropriate *Layout*, when the *VideoCard* context is not activated. It 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 methodology.

B. Observations on Complexity

In the algorithm 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 the same phenomenon as handling constraints after computing the array, which is known to be intractable [15].

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 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 [24]. Let N_f be the number of features and N_c the number of contexts. Assuming that both numbers are similar, we have:

TABLE VI
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
...	...			

$$O\left(log\left(N_f{+}N_c\right)\right) \simeq O\left(log\left(2N_f\right)\right) \quad (8)$$

$$O\left(log\left(2N_f\right)\right) = O\left(log\left(2\right){+}log\left(N_f\right)\right) = O\left(log\left(N_f\right)\right) \quad (9)$$

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.

C. Computation time and optimizations

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

a) 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.

b) 2. Additional step based on propagation: An important step of SAT solvers is Boolean constraint propagation (BCP) [25]. 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 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. [15] proposed an optimization which exploits BCP. Let inf_{V_1}, ..., 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 is 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 VII shows the number of propagations and the number of propagated values with or without the first optimization, when applied to our case study. It shows that these optimizations affect two largely separate aspects. Indeed, the number of propagated values is almost the same with or without the first optimization. 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 VII
IMPACT OF THE PRE-PROCESSING OPTIMIZATION ON THE NUMBER OF PROPAGATIONS AND PROPAGATED VALUES.

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

Table VIII illustrates how each of these optimizations improve the average computation time for 30 executions of the CIT algorithm on our case study. The pre-processing of core/dead contexts and features drastically prunes the set of features and corresponding constraints. The first optimization

TABLE VIII
IMPACT OF THE OPTIMIZATIONS ON THE COMPUTATION TIME

	Computation time (seconds)
Without optimization	17.2
Optimization 1	4.6
Optimization 2	14.6
Both optimizations	**2**

reduces the computation time by 73%, the second one decreases it by (only) 15%. The two optimizations combined reduce the time of computation by an average of 88%.

To answer **RQ1**, we adapted a FBCOP system and proposed a test scenario generation algorithm. Its computation time is 2 seconds when applied to our case study, which highlights its lightweight nature.

V. TEST SUITE REARRANGEMENT

As discussed in Section III-B, by decreasing the creation cost the effort of creating a generated test suite can be minimised. The creation cost of a test suite was defined as the number of context switches (i.e., activations or deactivations) needed to simulate all scenarios from the suite. In this Section, we answer **RQ2** by proposing a test suite rearrangement algorithm which minimizes this cost.

We first define the *distance* between two test scenarios to be the number of activated (resp. deactivated) contexts in the first scenario which get deactivated (resp. activated) in the second one. Thus, the distance between two test scenarios is the number of context switches needed to go from one configuration of the contexts 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.

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 $L$
```

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, for the test suite shown in Table IX, the rearranged test suite has a creation cost of 89 instead of 161 as for the original test suite which is a decrease of 45%. 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 44% in cost, with a variance of 4.5% over all executions of the CIT algorithm.

TABLE IX
THREE TEST SCENARIOS OF A RE-ARRANGED TEST SUITE

Scenario	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 interactions that are responsible for the failures. When a failure occurs in a given test scenario, it is highly likely that the failure is caused by an interaction with a new feature added in this test scenario. The goal of the rearrangement is to reduce the number of context switches. An indirect consequence is that it also drastically reduces the number of feature switches and hence the number of feature interactions to inspect.

Our answer to **RQ2** is thus a simple-to-implement rearrangement algorithm which greedily tries to minimize the creation cost of a given test suite, a metric closely related to the effort of simulating and creating this test suite.

VI. TEST SUITE AUGMENTATION

Like any software system, FBCOP systems are subject to continuous development that eventually expands the initial system's range of possible behaviours and contexts. 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 without recomputing the entire test suite, as was discussed in Section III-C. Our answer to **RQ3** is then a greedy algorithm for incremental test suite augmentation that works in two steps:

a) 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 remains 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.

b) 2.: As second step, the algorithm checks if the augmented test suite is complete (i.e., complete coverage of the 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 A of the CIT algorithm is no longer empty. The generated test scenarios are then rearranged with the algorithm of Section V (updated test scenarios naturally keep their original order). The result is a complete augmented test suite.

A. 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 scenario, if this is possible.

a) Strategy 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.

b) Strategy 2: Test expansion 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 randomize 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. [15], we are not adding constraints after computing the whole array, but just checking compatibility for a restricted number of pairs.

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 now present an update procedure. 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).

B. Performance

We now evaluate the performance of the above strategies on our case study of Figure 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 to a user's age, corresponding to the orange part of Figure 1. It comprises contexts *Age*, *Teen* and *Adult* and features *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 Figure 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 Figure 4, 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.

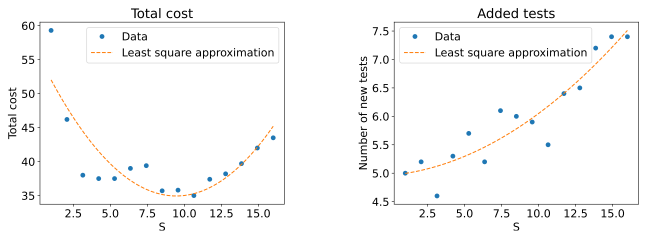

Fig. 4. Influence of parameter S on the total cost of augmenting a test suite

In Step 2 of Strategy 2, we combine partial scenarios to a maximum of S consecutive test scenarios to update them while minimizing 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 Figure 5 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 Figure 4.

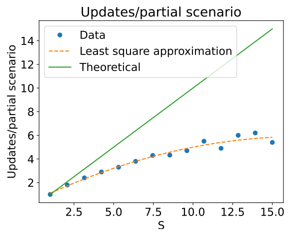

Fig. 5. Updates/partial scenarios, on average

We now compare both incremental augmentation strategies. Table X shows the costs of augmenting the test suite when moving from the original to the second version of the system. Table XI 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).

TABLE X
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

NOREUSE produces the least amount of test scenarios, which is expected since CIT's objective is to minimize 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 XI by more than a factor 4 the cost of creating a new test suite by reusing the previous one.

TABLE XI
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

VII. CONCLUSION AND FUTURE WORK

This paper addressed three main research questions, namely: **(RQ1)** How to generate a pertinent yet tractable set of test scenarios for a given FBCOP application? **(RQ2)** How to minimise the effort of creating these generated test scenarios? **(RQ3)** How to incrementally adapt a previously generated set of scenarios upon evolution of the application to a new version?

Our answer to RQ1 was to reduce 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 minimize the number of context switches needed to simulate the entire suite. Finally, to answer RQ3 we explored an algorithm to incrementally augment existing test suites upon program evolution.

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 systems if we obtain a set of contexts and features from them and discretize these contexts and features [26].

Validation of context-oriented programs is still at its infancy. There exist a wide range of possibilities to improve upon the work presented in this paper. We could improve Cohen's algorithm [15] by exploiting the fact that our SAT formulas are mostly produced from feature diagrams in FBCOP. Exploiting such representation with algorithms such as those presented by Johansen [27] and Kowal [28] could help identify dead/core features or other anomalies that should be embedded in or rejected from each test scenario. Another way would be to use Quantified Boolean Formula solvers [29]. 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 [30], [31] but have not yet been adapted to handle systems with constraints.

The rearrangement process could also be extended by considering other objectives. In the spirit of Devroey et al. [20], one could target rearrangements that prioritize 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.

Similarly, new ways to generate partial scenarios in the augmentation procedure could be considered.

The main objective of this paper being to pose the foundations of our new theory, the experimental section was reduced to its minimum. So far, our prototype has been applied to a small academic case study only. The cost of creating a test suite for it and augmenting it has been reduced by more than half. In future work, we hope to investigate how its performance scales to industrial-scale case studies.

Acknowledgements

Axel Legay and Pierre Martou are supported by the CYRUS project — Convention n° 8227, Pôle LiW — Appel 27.

References

[1] P. Costanza and R. Hirschfeld, "Language constructs for context-oriented programming: An overview of contextl," in *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*, p. 1–10, ACM, 2005.

[2] R. Hirschfeld, P. Costanza, and M. Haupt, "Generative and transformational techniques in software engineering ii," ch. An Introduction to Context-Oriented Programming with ContextS, pp. 396–407, Springer, 2008.

[3] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux, "Subjective-c: Bringing context to mobile platform programming," in *Proceedings of 3rd International Conference on Software Language Engineering*, SLE '10, pp. 246–265, Springer, 2011.

[4] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Javactx: Seamless toolchain integration for context-oriented programming," in *Proceedings of 3rd International Workshop on Context-Oriented Programming*, COP '11, pp. 4:1–4:6, ACM, 2011.

[5] B. Duhoux, K. Mens, and B. Dumas, "Implementation of a feature-based context-oriented programming language," in *Proceedings of the Workshop on Context-oriented Programming*, COP '19, pp. 9–16, ACM, 2019.

[6] B. Duhoux, K. Mens, B. Dumas, and H. S. Leung, "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)*, vol. 2510 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019.

[7] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008.

[8] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801 – 1817, 2012.

[9] N. Cardozo, S. Günther, T. DHondt, and K. Mens, "Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations," in *International Conference On Software Engineering Advances (ICSEA'11)*, pp. 130–135, IARIA, 2011.

[10] N. Cardozo, K. Mens, P.-Y. Orban, S. González, and W. De Meuter, "Features on demand," in *Proceedings of 8th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pp. 18:1–18:8, ACM, 2014.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.

[12] H. Hartmann and T. Trew, "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, pp. 12–21, IEEE, 2008.

[13] R. Capilla, O. Ortiz, and M. Hinchey, "Context variability for context-aware systems," *Computer*, vol. 47, no. 2, pp. 85–87, 2014.

[14] K. Mens, R. Capilla, H. Hartmann, and T. Kropf, "Modeling and managing context-aware systems' variability," *IEEE Software*, vol. 34, no. 6, pp. 58–63, 2017.

[15] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[16] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Pairwise testing for software product lines: comparison of two approaches," *Software Quality Journal*, vol. 20, no. 3, pp. 605–643, 2012.

[17] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012.

[18] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Multi-objective test generation for software product lines," in *Proceedings of the 17th International Software Product Line Conference*, pp. 62–71, 2013.

[19] R. M. Hierons, M. Li, X. Liu, J. A. Parejo, S. Segura, and X. Yao, "Many-objective test suite generation for software product lines," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–46, 2020.

[20] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P. Schobbens, and P. Heymans, "Statistical prioritization for software product line testing: an experience report," *Software & Systems Modeling*, vol. 16, no. 1, pp. 153–171, 2017.

[21] S. Lity, M. Al-Hajjaji, T. Thüm, and I. Schaefer, "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, 2017.

[22] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *Software & Systems Modeling*, vol. 18, no. 1, pp. 499–521, 2019.

[23] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *11th International Software Product Line Conference (SPLC '07)*, pp. 23–34, IEEE, 2007.

[24] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[25] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[26] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[27] M. F. Johansen, O. Haugen, and F. Fleurey, "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, p. 46–55, ACM, 2012.

[28] M. Kowal, S. Ananieva, and T. Thüm, "Explaining anomalies in feature models," *ACM SIGPLAN Notices*, vol. 52, no. 3, p. 132–143, 2016.

[29] J. Mauro, "Anomaly detection in context-aware feature models," in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21, pp. 1–9, 2021.

[30] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of SAT solutions for configurable systems: Are we there yet?," in *12th IEEE Conference on Software Testing, Validation and Verification, (ICST '19)*, pp. 240–251, IEEE, 2019.

[31] E. Baranov, A. Legay, and K. S. Meel, "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)*, pp. 1114–1126, ACM, 2020.