JENRS

Received: 17 March 2022, Revised: 28 April 2022, Accepted: 10 May 2022, Online: 25 May 2022

DOI: https://dx.doi.org/10.55708/js0105019

Evolution in Software Product Lines: Defining and Modelling for Management

Amougou Ngoumou *, Marcel Fouda Ndjodo

Department of Computer Science, Higher Teacher Training College, University of Yaounde I, Cameroon
*Corresponding author: Amougou Ngoumou, PO Box 47, Yaounde, Cameroon, Tel.: (237) 677 755 148, Email: ngoumoua@yahoo.fr

ABSTRACT: Evolution in Software Product Line (SPL) is claimed when there are changes in the requirements, product structure or the technology being used. Currently, many different approaches have been proposed on how to manage SPL assets and some also address how evolution affects these assets. However, the usefulness, effectiveness and applicability of these approaches are unclear, as there is no clear consensus on what an asset is. In this work, we plan to reduce complexity in SPL evolution management. For this goal, the difficulty is defining and modeling SPL evolution and we expect to propose a flexible way to manage it. However, a large variety of artifacts is considered in SPL evolution studies, but feature models are by far the most researched ones. Feature models are widely used to represent SPLs and have been greatly developed in the Feature-Oriented Reuse Method (FORM). Consequently, in our previous works, after observed that this method has a loose structure since it does not provide guidance to reuse and rigorously analyze its assets, we have extended FORM to FORM/BCS (the Feature Oriented Reuse Method with Business Component Semantics) by enveloping its assets among which feature models with business component semantics. The contribution and the novelty of this work is that, by highlighting formally the concept of software asset and revisiting feature business components, to add new information when analyzing a domain, such as clashing actions. conflicts or undesired interactions between existing features in a product line and new features due to evolution of the product line can be manage in a flexible way.

KEYWORDS: Evolution, Software Product Line, feature-orientation, domain analysis, business components, reuse

1. Introduction

The Software Product Line Engineering (SPLE) [1] is an approach that aims at creating individual software applications based on a core platform, while reducing the time-to-market and the cost of development [2]. Many SPLE-related issues have been addressed both by researchers and practitioners, such as variability management, product derivation, reusability, etc. According to authors in [3], a Software Product Line (SPL) is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". The main benefit of defining a SPL is that the reuse of all assets can be systematically organized [4].

There are two distinct phases in SPL definition: domain engineering and application engineering. The domain engineering phase starts with domain analysis, where domain knowledge is used to identify common and variable features, and these features are then realized during domain design and implementation. Application engineering focuses on product creation, first by identifying customer needs, which are then used to guide product derivation. In this way, the cost of developing and maintaining core assets is spread across all the products in a SPL, and is not specific to each separate product [5]. Note that the domain knowledge, asset realization, product configuration, etc., can all evolve over time [6].

The concept of evolution [7, 8] is intrinsic to software, since customer requirements and needs change over time, so software must evolve to remain useful [9]. However, the software evolution process is quite challenging since a fragile balance must be maintained: software quality must be preserved but software structure tends to

degrade over time. The following challenges have been identified [10, 6] in the case of SPL evolution: 1) there are different types of assets, which are defined at different levels of abstraction and variability; 2) there is a high number of interdependencies between assets; 3) a SPL usually has a longer life-span than a single product; and 4) a SPL is larger and more complex than its individual products. Currently, many different approaches have been proposed on how to manage SPL assets and some also address how evolution affects these assets. However, the usefulness, effectiveness and applicability of these approaches are unclear, as there is no clear consensus on what an asset is. In this work, our research method consist of highlighting formally the concept of software asset and revisiting feature business components, to add new information when analyzing a domain, such as clashing actions so that we can manage evolution in a flexible way. The base is feature models[11] which are widely used to present commonality and variability (C & V) information of a product line compactly (see Figure1. for example). We have extended Feature models in the Feature Oriented Reuse Method with Business Component Semantics (FORM/BCS) [12, 13, 14, 15, 16, 17]. Each product in the product line is derived from a selection of a valid combination of features [18] —a process known as product configuration [19, 20].

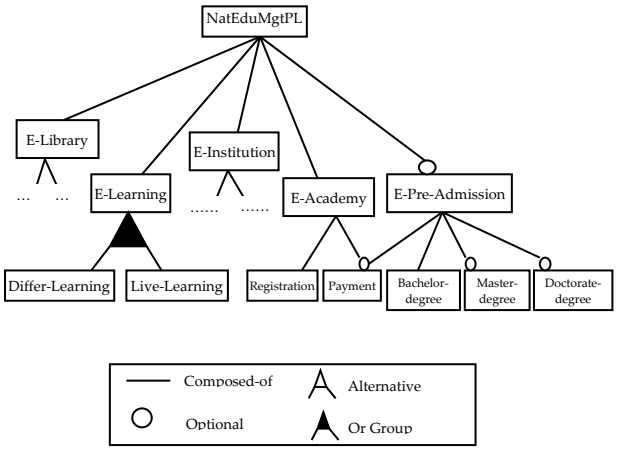

Figure 1. Partial feature model of NatEduMgtPL

Figure 1. presents an example of enterprise software for tertiary institutions of an anonymous country. The product line, referred to as National Educational Management Product Line (NatEduMgtPl), was initiated by the Ministry of Higher Education in that country. The vision of the product line is to provide software products to state universities, other higher institutions, and Enterprise Resource Planning (ERP) vendors. The educational institutions in the country implement the BMD (Bachelor, Master and Doctorate) system - which make their core operations largely the same- hence a product line.

The remainder of the paper is organized as follows. Section 2 details out research design, method, instrument and analysis technique. Section 3 highlights formally software assets and revisits FORM/BCS feature business components. Section 4 defines and models evolution in Software product Line so that we can see how evolution affects feature business components. Section 5 presents related work and section 6 concludes the work and gives perspectives.

2. Research design, method, instrument and analysis technique

Many different approaches have been proposed on how to manage SPL assets and some also address how evolution affects these assets. However, the usefulness, effectiveness and applicability of these approaches are unclear, as there is no clear consensus on what an asset is.

In this regard, we think that the first concern on evolution in SPL is to establish a clear vision on concepts and then processes. The envy to clarify software assets encourages us to first highlight formally this concept. To avoid lack of understanding and ambiguities, we specify the description of software assets using Z notation.

Secondly, knowing that the management of software product line evolution is complex and this evolution is due to requirements and needs change, we revisit the specification of feature business components proposed in the FORM/BCS method [12, 13, 14, 15, 16, 17], as it is the first software asset produced when analyzing the domain, to anticipate evolution very early. In this revision, we enrich features business components with new information such as clashing actions so that we can manage evolution in a flexible way. In the proposed analysis technique, for feature business components, the analyst must find and give, if it's possible, a clash action for all actions in that asset. These clashing actions advice on conflicts and undesired interactions between features and the analyst can avoid or correct them when new features and adaptation points due to evolution appear in user's requirements and needs.

We know that SPL is actually a continue process and we cannot think about all possible variant, but, by this contribution, we want to improve the flexibility of that process.

3. Software Assets

A software asset is composed of a set of software products derived from different activities of the life cycle. Specifically: requirements, architecture definition, analysis model, design model, code, test programs, test reports.

The different products which compose a software asset are in fact the representation of that asset at different level of abstraction (need, analysis, design, realization, texts). When the software asset is reused, each of these software assets can then be reused in the corresponding step (before, during and after coding). Specifically, test programs are strongly reusable. The person who desires

evaluate a software asset for reuse can take existing test programs to enforce the software asset in his own environment. It is important not to limit reuse at code level, but exploit all software assets.

Reusable software assets must be provided with necessary information for their reuse (the software asset description, also call « meta-information » [21]). This additional information allows facilitate software asset manipulation during his life cycle. It is in particular following elements: Classification information which allows facilitate corresponding software assets research, description of software asset which allows to understand rapidly functions and main features of the software asset, documentation of the software asset which allows to understand how enforce and customize the software asset, information related to tests and software asset qualification to facilitate his evaluation by a potential reuse stakeholder, information about software asset origin and property to obtain support or complementary information.

All these characteristics are summarised in the specification below using Z notation.

Table 1: Specification of Software Assets

SoftwareAsset = = [identifier: **TEXT**;
is_composed_of: 𝔽 **SoftwareAsset**
uses: 𝔽 **SoftwareAsset**
description: **Description**
body: **Body** |]

This schema in Table 1 shows that a software asset is made up of two types of information: the **body** (containing effectively reuse software assets) and **description** (containing information allowing reuse process support). Information of qualification and classification correspond respectively to the qualification process and the classification process.

This model also brings to light the **imbrications** of software assets, and the fact that, beside composition relations, software assets can have others types of links illustrating, for example, the fact that a software asset uses an other software asset. That means, a software asset needs, to run, functionalities of another software asset. The software asset **reuser** must then decide if he also reuses associated software assets or he is able to provide himself an equivalent implementation. Typically, a vertical software asset, if it has an important granularity, will lean probably on component techniques (for example graphical objects or a middleware).

3.1 Software Asset Description

The description of a software asset gives its intention, the engineering activity the descriptor plans to perform, its target, the concerned business and the environment that is the context. The above Z notation schema specifies software asset description.

Table 2: Specification of Software Asset Descriptions

Description = = [intention : **EngineeringActivity** ;
target : **Business;**
environment: **Context** |]

EngineeringActivity = =
AnalysisActivity | **DesignActivity** | **ImplementationActivity**
AnalysisActivity = {analyze, …}
DesignActivity = {design, decompose, describe, specify, …}
ImplementationActivity = {implement, …}

Business = = [domain: **Domain**; processes: 𝔽 **Process** |]

Details on the following concepts: Domain, Process, Business Activity, Context & Context-awareness can be found in [16].

3.2 Software Asset Bodies

A body of a software asset is composed of software products effectively reuse. These software products can be analysis models, design models, source codes, user documentation, runnable codes, test reports, test scenarios, test programs. The following schema models software asset bodies.

Table 3: Specification of Software Asset Bodies

Body = = **AnalysisModel** | **DesignModel** |
SourceCode | **UserDocumentation** |
RunnableCode | **TestReport** |
TestScenario | **TestProgram**

If we use the feature oriented reuse method with business component semantics, the body will be a feature realization if we are in the analysis stage, a conceptual realization, a process realization or a module realization if we are in the design stage.

4. Evolution in Software Product Lines

Feature models are widely used to represent SPLs and have been extended in the Feature Oriented Reuse Method with Business Component Semantics (FORM/BCS). Software product line evolution is the necessity to have in that product line new features, variability points or the death of old ones. This continuous phenomenon is due to changes in the requirements, product structure and newly emerging technologies. The integration of new features or variability points can creates conflicts or undesired interactions between them. For example when you add new features, they can enter in conflict with old ones. Let us take an example in libraries, if you want to *ensure a sufficient availability* of books and previously you *authorize long term loans*, the two features will be in conflict.

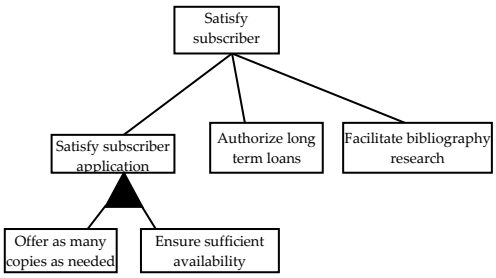

Figure 2. Feature model with a conflict

Equally, when you remove an old feature, if this feature is used by another one, you will create inconsistency. That why the management of this situation is complex. To study evolution in SPLs, we first look at the feature business component which is a software asset in which the body is a feature realization [12] to see how we can improve the specification of his constituents that are his description and body.

Table 4: Specification of Feature Business Components

FeatureBusinessComponent = = [identifier: **TEXT**;
is_composed_of: 𝔽 **FeatureBusinessComponent**
uses: 𝔽 **FeatureBusinessComponent**
description: **Description**
body: **FeatureRealization** |]

Knowing that processes are essentials in the description of feature business components, we start by revising their specification.

4.1 Processes with clashing actions

Evolution can occur in requirements or in new technologies and the first thing to observe is that, when a new variation point appears, to take it into consideration, we must guaranty that it don't create conflict with an existing feature or an undesired interaction between features in the product line. We think that, to avoid these conflicts, it is useful to anticipate them when analyzing a domain. We introduce then new information such as clashing actions when modeling processes.

Table 5: Specification of Processes

Process = = [actions: 𝔽 **BusinessActivity**;
clashingactions: 𝔽 **BusinessActivity**;
input-elements : 𝔽 **BusinessObjects** ;
output-elements : 𝔽 **BusinessObjects** ;
precision : **Precision** |]
BusinessObjects = = 𝔽 **Class**
Class = = [name: **Name**; attributes : 𝔽 **Attribut**;
operations : 𝔽 **Operation** |]
Precision
Name
Attribute
Operation

4.2. Specifying clashing tasks in business activities

To manage evolution in software product lines, it is important to decompose business activities so that we can detect antagonist tasks between them. Antagonist tasks are tasks which cannot be performed together. A business activity has a set of "mandatory" tasks, a set of "optional" tasks, a set of "alternative" tasks, a set of "or" tasks and a set of "clashing" tasks. It can be primitive or not. The following schema specifies business activities for the management of evolutions.

Table 6: Specification of Processes

BusinessActivity == [name: **Name**;
decomposition: [mandatory: 𝔽 **BusinessTask** ;
optional: 𝔽 **BusinessTask**;
alternative: 𝔽 𝔽 **BusinessTask**;
or: 𝔽 𝔽 **BusinessTask]**;
clashing: 𝔽 **BusinessTask;**
primitive: **Logic** |]

When the context is clear we write:
mandatory (ba) for *mandatory (decomposition(ba))*
optional(ba) for *optional(decomposition(ba))*
alternative(ba) for *alternative(decomposition(ba))*
or(ba) for *or(decomposition(ba))*
decomposition (ba) for mandatory (ba) $\cup$ *optional (ba)* $\cup$ ($\cup$ *(A* $\in$ *alternative(ba)))*

We say that a business activity *ba* is abstract if *decomposition (ba)* = $\varnothing$.

We define the set
Abstract_Business_Activity = {ba:Business_Activity • *decomposition (ba)* = $\varnothing$}

4.3 Business tasks

The decomposition of tasks allows detecting antagonist tasks. A business task has a set of "mandatory" operations, a set of "optional" operations, a set of "alternative" operations, a set of "or" operations and a set of "clashing" operations. It can be primitive or not. The following schema specifies business tasks for the management of evolutions.

Table 7: Specification of Processes

BusinessTask == [name: **Name**;
decomposition: [mandatory: 𝔽 **BusinessOperation** ;
optional: 𝔽 **BusinessOperation**;
alternative: 𝔽 𝔽 **BusinessOperation**;
or: 𝔽 𝔽 **BusinessOpeartion**];
clashing: 𝔽 **BusinessOperation;**
primitive: **Logic** |]

In a similar manner, when the context is clear we write:
mandatory (bt) for *mandatory (decomposition(bt))*
optional(bt) for *optional(decomposition(bt))*
alternative(bt) for *alternative(decomposition(bt))*
or(bt) for *or(decomposition(bt))*

decomposition (bt) for mandatory (bt) ∪ optional (bt) ∪ (∪ (A ∈ alternative(bt)))

We say that a business task *bt* is abstract if *decomposition (bt) = ∅.*

We define the set

Abstract_Business_Task = {bt:Business_Task • decomposition (bt) = ∅}

4.4 Evolution management functions

4.4.1. Basic functions

The decomposition of business activities and business tasks allows defining evolution management basic functions:

- *run* which given two operations return *failure* if the two operations cannot be run together in the same system or *success* if they can be.
- *clashingtasks* which given a business task provides the set of his clashing tasks.
- *conflictactivities* which given a business activity provides the set of his conflict business activities

run: BusinessOperation x BusinessOpeartion ↔ {failure, success}

clashingtasks: BusinessTask ↔ 𝔽 BusinessTask
∀bt1, bt2: BusinessTask, bt2 ∈ clashingtasks(bt1)⇔
∃(bo1, bo2) ∈ operations(bt1) x operations(bt2) •
run(bo1, bo2) = failure

conflictactivities: BusinessActivity ↔ BusinessActivity
∀ba1, ba2: BusinessActivity, ba2 ∈ conflictactivities(ba1)⇔
∃(bt1, bt2) ∈ tasks(ba1) x tasks(ba2) •
bt2 ∈ clashingtasks(bt1)

4.4.2 Evolution mechanism

The specification of processes (sub section 2.1.2) shows that a process can be seen as a set of business activities. A non primitive business activity has decomposition. This decomposition groups the set of his "mandatory" tasks, the of his "optional" task, the of his "alternative" tasks and the set of his "or" tasks. A business activity has also a set of "clashing" tasks. A clashing task of a business activity is a task which cannot run with the tasks in his decomposition.

In a software product line, evolution is the apparition of a new variation point or the disappearing of an old one. A new variation point in feature business component as specified in the Feature Oriented Reuse Method with Business Component Semantics is a new feature with his variation points. A feature corresponds to a business activity [12]. To consider a new variation point, we must check if this new variation point doesn't create a clash with the existing ones.

Each new adaptation point has a parent feature and the evolution process of a feature business component consists of inserting the new feature as part of his parent. From there, we define the two following functions which are essential in our evolution mechanism: *is_clashed* and *insert*.

Given a feature business component *fbc* and his new feature adaptation point *nap*, the function *is_clashed* returns "false" if for each feature in the solution part of *fbc*, the activity of *nap* is not in conflict with the activity of *f*.

Given a feature business component *fbc* and his new feature adaptation point *nap*, the function *insert* returns the feature business component *fbc* containing the new feature adaptation point *nap* as an adaptation point.

is_clashed: FeatureBusinessComponent × FeatureAdatationPoint ↔ Boolean
∀ fbc : FeatureBusinessComponent, fap: FeatureAdaptationPoint,
is_clashed (fbc, fap) = false ⇔ ∀ f ∈ decomposition(realization(fbc)),
activity(feature (nap)) ∉ conflictactivities (activity (f)))
is_clashed(fp, ci) = true ⇔ ¬ (∀ f ∈ decomposition(realization(fbc)),
activity(feature (nap)) ∉ conflictactivities (activity (solution (fbc))))

insert: FeatureBusinessComponent × FeatureAdaptationPoint ↔ FeatureBusinessComponent ∀ fbc : FeatureBusinessComponent, fap : FeatureAdaptationPoint,
insert(fbc, fap) = fbc • fap ∈ decomposition(solution (realisation(fbc))) ∧
fap ∈ adaptationpoints(solution(realization (fbc)))]

Given a feature business component *fbci* and a finite set of new adaptation points *NAP*, the evolved feature business component *fbco* is obtained following the algorithm below:

Algorithm: Evolution management
Result: fbco: FunctionalPerspective
fbci: FunctionalPerspective ;
NAP: 𝔽AdaptationPoint;
wfbc: FunctionalPerspective;
For each nap **in** NAP
  If no_clash (fbci, nap) **then**
    wfbc:= insert(fbci, nap);
  else
    Write ("FAILURE")
  end
end
fbco:= wfbc

5. Related Works

Stability is one of the most important properties of software. It is defined as "The capacity of the software product to avoid unexpected effects from modification of the software" [22]. Many product line approaches assume that activities in domain and application engineering can take a fairly stable product line for granted. However, real-world product lines inevitably and continuously

evolve. Managing evolution is thus success-critical, particularly in model-based approaches to ensure consistency after changes to meta-models, models, and actual artifacts. In [23, 24], several authors have stressed the importance of approaches for product line evolution to avoid the erosion of a product line, i.e., the deviation from the product line model up to the point where key properties no longer hold. Several approaches have been proposed for managing the evolution of software product lines [4], ranging from verification techniques to ensure consistent evolution, to model-based frameworks dedicated to the evolution of feature-based variability models [25]. For example, an interesting research thread proposes evolution templates for co-evolving a variability model and related software artifacts [26, 27, 28].

A model-driven product line approach that focuses on the issue of domain evolution and product line architectures is described in [29]. Authors discuss several challenges for the evolution of model-driven software product line architectures and present their solution for supporting evolution with automated domain model transformations. Such transformations could also be useful in our context to realize the update rules to support the evolution of the variability models in SPLs when applying model-driven techniques.

Another example is the work in [30], who present tool support for the evolution of software product lines based on the grow-and-prune model. They support identifying and refactoring code that has been created by copy and paste and which might be moved from product level to product line level. Refactoring of a SPL is not the scope of our work which, for the moment, is not situated at the code level. However, the work and tool are useful to support refactoring the SPL code.

A SPL evolution approach that preserves the original behaviour of evolving product lines, *i.e.*, products that could be generated before evolution can still be generated after the evolution, is proposed in [31]. This of course is only possible if restricting the removal of certain needed features, which makes the process easier but also constitutes a limitation of this approach.

To keep a configuration consistent with a feature model even after evolution of the latter, in [32] authors present an approach that automatically evolves the configuration with respect to the changes performed in the model while also taking into consideration the possible cardinalities. Such an approach is useful.

Hyper feature models are introduced in [33]. These models are capable of versioning the features and their constraints to maintain evolution traceability over time and guarantee the compatibility of one version of a feature with versions of another one. Feature traceability is thus a central concern in SPL evolution approaches, and has been shown to be essential in a feature-oriented project [34]. In [35], authors was largely inspired by this earlier work on evolving software product lines, and extended this work by considering runtime management of such evolution.

Ideas developed in this contribution enter in pioneer works on feature orientation and come from our previous articles [12, 13]. The specificity of our approach is that, by putting inside feature business components, information able to guide evolution, we give intrinsic ability, which is since his genesis, to software product lines to evolve smoothly.

6. Conclusions and Future Research

Real-world product lines inevitably and continuously evolve, then we cannot avoid the necessity of evolution in a software product line. The scientific community tries to manage evolution in software product lines but faces some difficulties link to the definition and modeling of this phenomenon in software product lines. We think that this situation is due in a large part to the fact that there is no consensus on what a software asset is. In this article after defining formally what a software asset is, we have study evolution in the first software product line asset of the feature oriented reuse method with business component semantics, the feature business component. The result is that, we find and introduce new properties in the definition of processes such as clashing actions. These new fields have allowed defining news functions for the management of evolution. This work increases the ability of software product lines to evolve in a flexible way. We plan to study erosion of a software product line which is the deviation from the product line model up to the point where key properties no longer hold.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

We thank the Cameroonian Ministry of Higher Education for inspiring the feature model example given in this paper.

References

[1] K. Pohl, G. Böckle, F. J. van Der Linden, "Software Product Line Engineering: Foundations, Principles and Techniques," Springer Science & Business Media, 2005, doi.org/10.1007/3-540-28901-1.

[2] A. Benlarabi, A. Khtira, B. El Asri, "Learning to Support Derivation of Adaptable Products in Software Product Lines," Journal of Computer and Communications, vol. 8, pp. 114-126, 2020, doi.org/10.4236/jcc.2020.84009.

[3] P. Clements, L. Northrop, "Software Product Lines: Practices and Patterns," Addison-Wesley Professional, 2001.

[4] M. M. Samary, J. Simmonds, P. O. Rossel, M. C. Bastarrica, "Software Product Line Evolution: a Systematic Literature Review," Information and Software Technology, 2019, doi: 10.1016/j.infsof.2018.08.014.

[5] I. Gorton, "Essential Software Architecture," Springer, 2006.

[6] L. Montalvillo, O. Díaz, "Requirement-driven evolution in software product lines: A systematic mapping study," Journal of Systems and Software, vol. 122, pp. 110–143, 2016, doi:10.1016/j.jss.2016.08.053.
[7] M. Nieke, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, I. Schaefer, "Guiding the evolution of product-line configurations," Software and Systems Modeling, Springer, 2021, doi.org/10.1007/s10270-021-00906-w.
[8] D. Hinterreiter, L. Linsbauer, K. Feichtinger, H. Prähofer, P. Grünbacher, "Supporting feature-oriented evolution in industrial automation product lines," Concurrent Engineering: Research and Applications, vol. 28, no. 4, pp. 265–279, 2020, doi.org/10.1177/1063293X20958930.
[9] I. Sommerville, "Software Engineering," 10th, Addison Wesley, 2015.
[10] G. Botterweck, A. Pleuss, "Evolving Software Systems," In: ed. by T. Mens, A. Serebrenik, A. Cleve, Springer, Chap. Evolution of Software Product Lines, pp. 265–295, 2014.
[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Pennsylvania , USA, 1990.
[12] M. Fouda, N. Amougou, "The Feature Oriented Reuse Method with Business Component Semantics," International Journal of Computer Science and Applications, vol. 6, no. 4, pp. 63-83, 2009.
[13] M. Fouda, N. Amougou, "Product Lines' Feature-Oriented Engineering for Reuse: A Formal Approach," International Journal of Computer Science Issues, vol. 7, no. 5, pp. 382-393, 2010, doi:10.1.1.402.5014.
[14] M. Fouda, N. Amougou, "Transformational Variability Modelling Approach To Configurable Business System Application," in Software Product Line – Advanced Topic, Edited A. O. Elfaki, Intech Publisher, pp. 43-68, 2012, doi: 10.5772/37776.
[15] N. Amougou, M. Fouda, , "Feature-Relationship Models: A Paradigm for Cross-hierarchy Business Constraints in SPL," International Journal of Computer Science and Information Security, vol. 16, no. 9, pp. 112-124, 2018.
[16] N. Amougou, M. Fouda, , "Context metamodel in pervasive systems for dynamic software product lines," Journal of Software Engineering & Intelligent Systems, vol. 5, no. 3, pp. 124-137, 2020.
[17] N. Amougou, M. Fouda, , "Extended dynamic software product lines architectures for context integration and management," Journal of Software Engineering & Intelligent Systems, vol. 6, no. 1, pp. 28-41, 2021.
[18] A. Z. Umar, J. Lee, "A Model-Based Approach to Managing Feature Binding Time in Software Product Line Engineering," In: MODELS 2018 Workshops, Octobre, Denmark, 2018.
[19] G. P. Espinel-Mena, J. L. Carrillo-Medina, M. Flores-Calero, M. Urbieta, "Software Configuration Management in Software Product Lines: Results of a Systematic Mapping Study," IEEE Latin America Transactions, vol. 20, no. 5, pp. 718-730, 2022, doi: 10.1109/TLA.2022.9693556.
[20] T. Kehrer, A. Schultheiß, T. Thüm, P. M. Bittner, "Bridging the Gap Between Clone-and-Own and Software Product Lines," IEEE, 2021, doi: 10.1109/ICSE-NIER52604.2021.00013.
[21] K. Even-André, "Software Reuse: A Holistic Approach," John Wiley & Sons, 1995.
[22] B. Kitchenham, S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Technical report no. EBSE-2007-01, Keele University, Keele, UK, 2007, doi:10.1.1.117.471.
[23] S. Deelstra, M. Sinnema, J. Bosch, "Variability assessment in software product families," Information and Software Technology, vol. 51, no. 1, pp. 195–218, 2009, doi.org/10.1016/j.infsof.2008.04.002.
[24] S. Johnsson, J. Bosch, "Quantifying software product line ageing," In: Limerick, Ireland, pp. 27–32, 2000.
[25] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, S. Kowalewski, "Model-driven Support for Product Line Evolution on Feature Level," Journal of Systems and Software, vol. 85, no. 10, pp. 2261–2274, 2012, doi.org/10.1016/j.jss.2011.08.008.
[26] C. Seidl, F. Heidenreich, U. Aßmann, "Co-evolution of Models and Feature Mapping in Software Product Lines," In: ACM, Salvador, Brazil, pp. 76–85, 2012, doi.org/10.1145/2362536.2362550.
[27] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, P. Borba, "Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel," In: ACM, Tokyo, Japan, pp. 91–100, 2013, doi.org/10.1145/2491627.2491628.
[28] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, L. Kulesza, "Safe Evolution Templates for Software Product Lines," Journal of Systems and Software, vol. 106, pp. 42–58, 2015, doi.org/10.1016/j.jss.2015.04.024.
[29] G. Deng, D. C. Schmidt, A. Gokhale, J. Gray, Y. Lin, G. Lenz, "Evolution in model-driven software product-line architectures," In: P. Tiako, ed. Designing Software-intensive SystemsIdea Group Inc. (IGI), pp. 1280–1312, 2008.
[30] T. Mende, F. Beckwermert, R. Koschke, G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," In: IEEE CS, pp. 163–172, 2008, doi: 10.1109/CSMR.2008.4493311.
[31] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, P. Borba, "Investigating the Safe Evolution of Software Product Lines," In: ACM, Portland, Oregon, USA, pp. 33–42, 2011, doi.org/10.1145/2189751.2047869.
[32] N. Gamez, L. Fuentes, "Software Product Line Evolution with Cardinality-Based Feature Models," In: Springer Berlin Heidelberg, Pohang, South Korea, pp. 102–118, 2011, doi: 10.1007/978-3-642-21347-2_9.
[33] C. Seidl, I. Schaefer, U. Aßmann, "Integrated Management of Variability in Space and Time in Software Families," In: ACM, Florence, Italy, pp. 22–31, 2014, doi.org/10.1145/2648511.2648514.
[34] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, J. Guo, "Feature-oriented Software Evolution," In: ACM, Pisa, Italy, vol. 17, pp. 1–8, 2013, doi.org/10.1145/2430502.2430526.
[35] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, C. Schumayer, "Evolution in Dynamic Software Product Lines," Journal of Software: Evolution and Process, John Wiley & Sons, Ltd., 2020, doi:10.1002/smr.2293. hal-02952741v2.

Amougou Ngoumou is a Senior Lecturer of computer science at the Department of Computer Science of the Higher Teacher Training College of the University of Yaounde I (Cameroon). He has received Bachelor of Computer Science (1998), Master of Computer Science (2001) and PhD degree in Computer Science (2011) at the University of Yaounde I (Cameroon).

His main research interests include Software Product Lines, Domain-Specific languages and Information Systems.

Marcel Fouda Ndjodo is a full professor of computer science and the Head of the Computer Science Department of the Higher Teacher Training College of the University of Yaounde I (Cameroon). He has received a PhD in Computer Science at the University of Aix-Marseille II (France, 1992).

He coordinates besides the information systems and numerical technologies of education at the higher teacher training college. He is author of many scientific publications and has supervised many PhD thesis in information systems and software engineering.