Journal of King Saud University – Computer and Information Sciences (2016) **28**, 364–380

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Change impact analysis for software product lines

Jihen Maâzoun [a,*], Nadia Bouassida [a], Hanêne Ben-Abdallah [b]

[a] *MIR@CL Laboratory, Sfax University, Tunisia*
[b] *Abdulaziz University, Jeddah, Saudi Arabia*

Received 5 June 2015; revised 9 January 2016; accepted 13 January 2016
Available online 30 March 2016

KEYWORDS

Software product line;
Feature model;
Model evolution;
Change impact management

Abstract A software product line (SPL) represents a family of products in a given application domain. Each SPL is constructed to provide for the derivation of new products by covering a wide range of features in its domain. Nevertheless, over time, some domain features may become obsolete with the apparition of new features while others may become refined. Accordingly, the SPL must be maintained to account for the domain evolution. Such evolution requires a means for managing the impact of changes on the SPL models, including the feature model and design. This paper presents an automated method that analyzes feature model evolution, traces their impact on the SPL design, and offers a set of recommendations to ensure the consistency of both models. The proposed method defines a set of new metrics adapted to SPL evolution to identify the effort needed to maintain the SPL models consistently and with a quality as good as the original models. The method and its tool are illustrated through an example of an SPL in the Text Editing domain. In addition, they are experimentally evaluated in terms of both the quality of the maintained SPL models and the precision of the impact change management.
© 2016 The Authors. Production and hosting by Elsevier B.V. on behalf of King Saud University. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Contents

1. Introduction . . . 365
2. Related work . . . 366
 2.1. SPL modeling . . . 366
  2.1.1. Feature model . . . 366
  2.1.2. Our UML profile for software product lines . . . 367

* Corresponding author.
E-mail addresses: jihenmaazoun@gmail.com (J. Maâzoun), Nadia.Bouassida@isimsf.rnu.tn (N. Bouassida), HBenAbdallah@kau.edu.sa (H. Ben-Abdallah).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2016.01.005
1319-1578 © 2016 The Authors. Production and hosting by Elsevier B.V. on behalf of King Saud University.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

2.2. SPL evolution levels and operations . . . 367
2.2.1. Feature model refactoring . . . 367
2.2.2. Feature model refinement . . . 368
2.2.3. Feature model arbitrary evolution . . . 368
2.3. Existing works on SPL change impact analysis . . . 368
2.4. Other SPL evolution issues . . . 368
3. Measuring the effort needed for evolutionary SPL change . . . 369
4. Feature model and design level change impact analysis . . . 370
4.1. Inter-feature model changes . . . 370
4.1.1. Add feature . . . 370
4.1.2. Remove feature . . . 370
4.1.3. Rename feature . . . 371
4.1.4. Move feature . . . 371
4.1.5. Split feature . . . 371
4.2. Intra-feature model changes . . . 371
4.2.1. Add element . . . 372
4.2.2. Remove element . . . 372
4.2.3. Rename element . . . 372
4.3. Impact classification . . . 372
5. Case study . . . 373
6. Evaluation . . . 374
6.1. SPL quality evaluation after evolution . . . 375
6.2. Change impact evaluation . . . 377
7. Conclusion . . . 378
References . . . 379

1. Introduction

A software product line (SPL) (Clements and Northrop, 2001) represents a set of software-intensive systems that share a common set of features and software assets pertinent to a specific application domain. Besides the common product features in a domain, an SPL also describes variability points that can be used to derive new products in the SPL domain. Thanks to the predictive and organized reuse of its features and software assets, the SPL promises decreased time to market and improved software development productivity.

Even though an SPL covers several software variants, the inevitable evolution of these latter induces an evolution of the SPL itself. Product variants evolve to meet new requirements introduced by new technologies, new business goals, or modified customer preferences. Such product variants' evolution feeds-back several types of changes on their SPL, e.g., the emergence of a new feature, disappearance of an obsolete feature, structural re-organization of a feature, etc. Managing the impact of product variants' evolution on the SPL must have a means to analyze the effects of a product variant change on the SPL in terms of change operations to conduct on all of the assets describing the SPL.

An SPL is often described in terms of a problem space and a solution space (Seidl et al., 2012). The problem space captures high-level requirements usually in the form of feature models, whereas the solution space contains shared assets like source code, design and test artifacts. Given the tight correlation between both spaces, any change induced by the SPL evolution must be managed in a consistent way in all pertinent assets. Most of the works dealing with SPL evolution, e.g., Pleuss et al. (2012), Passos et al. (2013), Seidl et al. (2012), Neves et al. (2011), Laguna and Crespo (2013), and Xue (2011), focus on the evolution of feature model-oriented SPL, but they do not address the impact of a change on the consistency of the various assets of the SPL, e.g., the design and the products' code. In addition, none of the existing works analyzes the cost of a change in terms of the effort estimated to handle the change; such change impact analysis is important, for instance, to examine the value added by a change.

Because features are more structured and coarse-grained than requirements, they facilitate the understanding and traceability of an SPL evolution (Passos et al., 2013). In fact, Passos et al. (2013) argue that changes ought to be managed in a feature-oriented manner. We agree with this argument since we believe that features can be the blueprints where evolution can be managed and from where it can be traced back to the design, code and other assets. Hence, managing SPL evolution implies, first, managing change at the problem space level (i.e., the feature model) and, then, tracing these changes to the solution space (i.e., the design).

To achieve this feature-oriented SPL evolution strategy, two questions must be addressed: how to keep the consistency between the feature model and the remaining assets, particularly the design? and how to measure the effort needed in the management of each change impact? To be addressed, both questions require an explicit specification of the relationship between the SPL feature model and its design. To do so, we use our previously proposed approach which extracts the feature model from source code and specifies it using a UML profile (Maazoun et al., 2013). Unlike existing SPL feature model extraction approaches (e.g., Acher et al. (2013), Lozano (2011), Ziadi et al. (2012), Al-Msie'Deen et al. (2012), and Paskevicius et al. (2012)), ours integrates the semantic aspect of the product variants. Moreover, it describes the SPL design with a UML profile that represents the SPL variation points enriched with information extracted from the feature model. The enrichment provides for the traceability between the feature model and the design.

Besides the explicit relationship between the feature model and design, SPL evolution management requires a means to identify the impact of each change operation. To meet this requirement, we first specified the different changes that may occur in a feature model when adding, removing, splitting, renaming or moving a feature or an element belonging to a feature. Secondly, we defined a set of rules that formalize the impact of each change while keeping the design diagrams and feature model consistent and maintaining traceability between the changed feature model and SPL design. In addition, to maintain the traceability between the changed SPL feature model and design, we adapted a set of semantic criteria that express linguistic relationships amongst affected elements names. The semantic relations are automatically determined either from the WordNet Dictionary (Ben-Abdallah et al., 2004) if they exist, or by measuring the similarity between the features' names and the design elements' names (classes, attributes and methods) thanks to the cosine distance (Salton and Buckley, 1988).

It is worth noting that, as reviewed in Botterweck and Pleuss (2014), SPL evolution has been examined in the literature from different perspectives: analysis of an SPL evolution history, planning of future SPL evolutions, and implementation of an SPL evolution. All of these perspectives have two common prerequisites: explicit specification of the evolutionary changes, and precise identification of their effects/impacts. The herein presented work contributes towards the satisfaction of these two prerequisites in order to manage (analyze the effects of and implement) changes on the SPL level, i.e., the SPL feature model and its design. In addition, compared to existing SPL evolution management approaches, our approach has the following merits: It manages the change impact on both the feature model and design assets; it maintains the traceability and consistency between these two assets; it uses a set of quantitative software metrics to help in assessing the efforts needed to manage a change—the assessment can be used to decide whether to accept a change request or refuse it; and it is highly automated through a tool support, named *Evo-SPL*, that manages the systematic evolution of software product lines. Besides automatically identifying all elements impacted by a change, Evo-SPL produces a report containing the number of additional changes required to ensure the consistency amongst the design diagrams and the feature model. The designer can use this report to decide about which changes should be rethought and/or canceled. Once, they decide to accept a change, Evo-SPL transforms automatically the design and feature model to make them consistent.

The remainder of this paper is organized as follows: Section 2 overviews the feature model concepts and existing works pertinent to SPL evolution strategies and change impact analysis and describes our UML profile adapted for SPL in terms of stereotypes, tagged values. Section 3 presents a set of metrics to measure the effort needed to manage a change impact. Section 4 presents the consistency of the evolution and the risks involved; in addition, it presents different inter and intra feature model changes, their impacts and the different consistency rules. Section 5 presents a case study of an SPL in the games application domain and its evolution management through our EVO-SPL tool. Section 6 evaluate SPL quality after evolution and change impact. Finally, Section 7 summarizes the presented work and outlines its extensions.

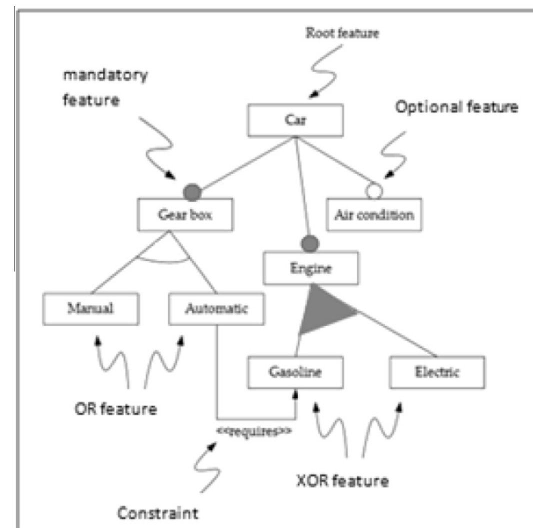


Figure 1 An example of a feature model.

2. Related work

In this section, we first review the feature model concepts and our UML profile used to specify software product lines. Secondly, in order to delimit the scope of the herein presented work, we highlight in Section 2.2 the main causes of SPL evolution and existing operations to handle them. In Section 2.3, we overview works dealing with SPL change impact analysis. Finally, in Section 2.4 analysis, we briefly discuss three main issues tightly linked to change impact analysis, mainly evolution consistency, risk assessment, and effort estimation.

2.1. SPL modeling

As mentioned in the introduction, an SPL is often described in terms of a problem space and a solution space (Seidl et al., 2012). The problem space captures high-level requirements usually in the form of feature models. The solution space contains shared assets like source code, design and test artifacts; in this paper, we consider that the solution space is a design modeled through our UML profile (Maazoun et al., 2014).

2.1.1. Feature model

Feature models are a popular means to express requirements in a domain at an abstract level. They are used to describe variable and common properties of products in a product line, and to derive and validate configurations of software systems. As introduced by the FODA method (Kang et al., 1990) and by Czarnecki and Eisenecker (2000a), a feature model represents a hierarchy of properties of domain concepts. A feature is a prominent or distinctive quality or characteristic of a software system or systems (Kang et al., 1990). It is a "distinguishable characteristic of a concept (e.g., component, system, etc.) that is relevant to some stakeholder of the concept" (Czarnecki and Eisenecker, 2000b).

Feature models allow designers to bridge the gap between the concrete code and the abstract information of documents such as the architecture and the design. Each feature model has a tree structure where each node represents a feature

(see Fig. 1). Feature variability is represented by the arcs and groupings of features. There are two different types of feature groups:

- Mandatory: (sub)features that must be present in every product in the line.
- Optional: (sub)features that may be present in some products.

In addition to the parental relationships between features, constraints between the nodes across the tree structure are used to constrain the derivation of a product from the SPL. The five most common cross-tree constraints are:

- And: all (sub)features must be selected together during the derivation of a product from the SPL.
- Xor: only one (sub)feature can be selected during the derivation of a product from the SPL.
- Or: one or more (sub)feature(s) can be selected during the derivation of a product from the SPL.
- Require: the selection of one (sub)feature necessitates the selection of the other.
- Exclude: two (sub)features cannot be part of the same product.

Fig. 1 illustrates an example of a fictitious feature model describing a car SLP where some features are mandatory (the "engine" and "gear box" features), and others are optional (the "air condition" feature). The "gear box" feature can be either manual or automatic, whereas the "engine" can be exclusively either "gasoline" or "electric". In addition, this SPL example specifies that if a car has an "automatic" gear box, then it must have a "gasoline" engine.

An SPL is reused to derive concrete products in the SPL domain. A concrete product is first defined by a *product configuration*, which resolves the variability by selecting or eliminating features in the feature model (Botterweck and Pleuss, 2014) while respecting the model's constraints. It is then developed based on the assets provided by the SPL. To this end, the feature model elements and its assets must be linked.

2.1.2. Our UML profile for software product lines

To ensure the traceability between the SPL feature model and its design, we propose to use our UML profile defined for SPL called *SPL-UML* (Maazoun et al., 2014). An SPL design is modeled in SPL-UML through the UML design diagrams (package, class, sequence, etc.) which are extended through a set of stereotypes introduced to model the variability aspect of software product lines. Due to space limitations, we next briefly review the seven stereotypes introduced in the class diagram:

- ≪optional≫ is used to specify optionality in a UML class diagram. The optionality can be associated with classes, packages, attributes or operations.
- ≪recommended≫ is used to specify recommendation in a UML class diagram. The recommendation stereotype applies to classes only.
- ≪mandatory≫ is used to specify obligatory elements. It can be associated with classes, packages, attributes or operations.
- ≪mandatory_association≫ is used to specify obligatory relation between classes in a UML class diagrams. It is graphically represented by a bold line.
- ≪optional_association≫ is used to specify optionality a relationship between classes in UML class diagrams. It is graphically represented by a dashed line.
- ≪Xor_association≫ is used to specify an alternative relation between classes in a UML class diagram.
- ≪Feature_Name≫ is used to specify the name of the feature to which the element belongs. It is pertinent to classes, packages, attributes and operations. It is a means of traceability between the SPL design and its feature model.

We note that, in terms of design, a feature can be either simple/elementary which contains only one design element like {package} and {class}, or composed of several design elements like {package, class}, {package, class, attribute, method}, etc.

2.2. SPL evolution levels and operations

Given the high cost of their development, SPL are set-up to have a long life-span and thus will evolve to cover new and modified requirements of their domains. Compared to single systems, SPL evolution has higher complexity due to the variability, the correlations between the problem and solution space models, and the inter-dependencies among the products in the line.

As discussed in the process framework for SPL evolution proposed by Botterweck and Pleuss (2014), an SPL evolution can be driven by business goals and external triggers for evolution (e.g., market changes), mismatches and suggested changes resulting from product derivation, and experiences with the products or SPL. In addition, depending on the change trigger, the framework of Botterweck and Pleuss (2014) classifies the SPL change at the product line level and/or the product level.

To handle SPL evolutionary changes at these levels for different purposes, several works dealt with SPL evolution in terms of feature model changes through four strategies: refactoring, refinement, specialization or arbitrary and general evolutions caused by change of requirements (Thüm et al., 2009).

2.2.1. Feature model refactoring

Refactoring restructures the feature model while preserving the same set of elements. It implies that no elements are added nor deleted. It was the focus of many researchers, e.g., Alves et al. (2006), Loesch and Ploedereder (2007), Mende et al. (2008), and Schulze et al. (2012). For instance, Alves et al. (2006) define a set of refactoring operations, like converting an "alternative" constraint to an "or"; these feature model refactoring operations restructure the SPL to improve it while preserving the behavior of its products. Mende et al. (2008) support refactoring of product variants that were created by copy and paste and which could be propagated to the SPL level. Schulze et al. (2012) propose variant-preserving SPL refactoring to ensure the validity of all SPL variants after refactoring. The objective of refactoring is to restructure the code of the SPL in the solution space within the context of feature-oriented programming.

2.2.2. Feature model refinement

Refinement preserves the set of products (no elements are deleted) while adding new features or deleting some constraints. Thus, the resulting feature model is a generalization of the original one, e.g., Borba et al. (2010) and Neves et al. (2011). The SPL refinement approach proposed by Borba et al. (2010) does not provide concrete operations for evolution changes. An improvement of this work was proposed by Neves et al. (2011) who propose templates for safe product line evolution. However, the steps described in the templates are manually performed. Furthermore, this work does not consider adding entirely new functionalities to the SPL.

2.2.3. Feature model arbitrary evolution

Within the arbitrary evolution perspective, Schubanz et al. (2013) and Pleuss et al. (2012) propose EvoPl, an approach that plans and manages long-term evolution of product lines. EvoPl allows the specification of historic and planned future evolutions in terms of changes at the feature model level. EvoPl has the advantage of reducing complexity through the use of model fragments to cluster related elements. The relationships between the fragments are represented with a model similar to the feature model named EvoFM.

Romero et al. (2013) propose *SPLEMMA*, a generic framework for controlled SPL evolution. SPLEMMA allows the validation of controlled SPL evolution by adopting a model-driven engineering approach. It has the advantage of capturing the evolution of an SPL independently of the kind of assets, technologies or feature models used for the product derivation. Authorized changes are described by the SPL maintainer and captured in a model used to generate tools that guide the evolution process and preserve the consistency of the whole SPL.

Also adopting a model-driven approach, Seidl et al. (2012) propose an approach for model-driven planning and monitoring of product line evolution. This approach treats evolution over the time and space dimensions. Within the time dimension, it models temporal concepts to support continuous evolution planning over a long period. Within the space dimension, it supports traces evolution from high-level decisions down to the implementation. This approach has two main limitations: it does not take into account the constraints between features, and the evolutionary change impact is handled only on traceability. For example, when dealing with the removal of a feature, the impact consists only in removing the traceability mapping without removing the code and the design corresponding to this feature. Consequently, the SPL code asset ends up containing a great number of code lines that will not be used in the derived products.

Finally, we note that existing works for SPL evolution need to model or detect the differences between models. This is done through model comparison or delta models (e.g., Haber et al. (2012) and Schaefer (2010)). Model comparison, where a model differencing algorithm is used (e.g., Xue (2011)), aims at comparing models or performing a difference between codes (e.g., EMFCompare[1]) to identify the changes that occurred to features. In other words, these works presume that evolutionary changes have occurred (either at the SPL level or product level) and they need to identify them. The herein presented work complements those works that identify changes at the product level and need to propagate them into the whole SPL. In addition, it supports those works that need to analyze the effects of requirements changes at the SPL level for several purposes like risk assessment, cost/effort analysis, implementation, etc.

[1] Eclipse-Foundation. EMF: Eclipse Modeling Framework 2.0. Website http://www.eclipse.org/modeling/emf/.

2.3. Existing works on SPL change impact analysis

Changes on an SPL affect the SPL models (feature model and assets), but they affect an existing product only if the product is re-derived, for instance to release a new version of the product that includes the changes made on SPL level. Change impact analysis is used to identify the changes incurred on the SPL models and, if needed, existing products that must be re-derived/reconfigured. It can be used to plan future evolutions of the SPL during its engineering and/or to handle new requirements triggered at the SPL level or product level.

To account for the effects of a change for SPL evolution analysis, planning and/or implementation purposes, traceability between the various SPL models is required. Traceability maps features to SPL design and implementation elements to enable feature-oriented product derivation and SPL evolution. Among the works interested in traceability, Passos et al. (2013), have envisioned a feature oriented project management and system development supporting traceability, feature oriented analysis of implementation artifacts, and feature oriented specific recommendation systems. Shen et al. (2009) propose a comprehensive feature oriented traceability model for SPL development, which provides mechanisms for various feature types. This framework offers visualized and comprehensive traceability representations for SPL development, throughout the four levels: goal model, feature model, feature implementation model and program implementations. Given the narrower scope of our work (SPL change impact management), we use our UML profile (see Section 2.1.2) which explicitly links design and feature model information in a simple yet comprehensive way.

The traceability information among the SPL feature model and its assets is the corner stone for SPL change impact analysis. In the scope of this paper, we consider change impact analysis that identifies all the elements affected by each change both at the feature model level and the associated assets. In other words, we do not deal with change impact analysis on product configurations derived from the SPL. In this context, very few works examined the impact of SPL evolutionary changes on the SPL models. For example, in Heider et al. (2012), variability change impact analyses can be automated using model regression testing trough an automated tool. This tool informs engineers about the impacts of variability model changes on existing products and re-derives all products and compares them with their previous version and reports the differences. Moreover, to support evolution, Cordy et al. (2012) propose a model-checking approach. Then, they propose a method to identify specific types of features and show that for such features, when added to an evolving SPL, only a subset of the products need to be model-checked again.

2.4. Other SPL evolution issues

Besides traceability, SPL evolution also must address the following issues tightly related to change impact analysis:

consistency of the adopted evolution, effort estimation in model evolution and risks involved in model evolution (Michalik and Weyns, 2011). Each of these issues can be dealt with either in the evolved SPL models only (feature model and assets), or also across the reconfigured derived products. As stated in the previous section, the scope of our work is limited to change impact management at the SPL models; hence, we will limit our discussion of these issues to the SPL models.

By evolution consistency, we mean that the evolved SPL models are internally (the feature model and each asset separately) inter-consistent (all models with respect to one another). In this context, Guo and Wang (2010) explores the possibility of consistency evolution of feature models from a perspective of atomic operations (e.g., add, remove and set a feature) and their semantics. In our approach, we propose an automated method that offers a set of recommendations to ensure the consistency of the SPL feature model and its design. The automation is provided by a precise definition of every change operation in terms of its pre-condition, post-condition and impact on both the feature model and the design. We note that our method is designed to be used during the implementation or planning of an SPL change, as opposed to during the SPL evolution history analysis.

For effort estimation in model evolution, Ramil and Lehman (2000) propose a suite of metrics for software evolution obtained from software change records. The author suggests that metrics are sufficient for effort estimation. In our work, we propose a set of new metrics adapted to SPL evolution and inspired from Chidamber and Kemerer (1994). The proposed metrics (see Section 3) provide for estimating the effort needed to manage the change impact both at the feature model level and across the SPL design. That is, the proposed metrics can be used to change impact in terms of number of elements (features, classes, attributes, methods, and packages) added or deleted.

Risk management aims to reduce potential risks and to offer opportunities for positive improvement in performance. In this context, Barry (1991) proposed a list of top ten risk categories. In a recent paper on risk management, the risk factors have been prioritized according to their frequency of occurrence and the impact that they possess (Shahzad and Iqbal, 2007), and thus a list of 14 risk factors with respect to their total impact has been identified. In Shahzad (2010), the authors propose a model that can be used to handle effectively the risk in the software development environment. In the context of SPL evolution, risks are the results from evolution and they can affect consistency, completeness and correctness. For consistency, when changes accumulate, related assets might be changed in different directions and no longer be compatible among themselves and/or with the feature model. Completeness is affected when given changes are applied to a large number of assets. An association could potentially be lost or rerouted during evolution, resulting in an asset being omitted from a configuration and blocked from further changes. Correctness is affected when changing an asset without propagating the change to the remaining assets/feature model. In our work, the pre-condition and post-condition defined for each change decrease the risks in terms of inconsistency and incorrectness of the evolved SPL models. In addition, completeness in our work is interpreted as completeness of the design with respect to the evolved feature model. To ensure the completeness of a changed SPL, we propose rules to manage the intra (feature model) and inter (feature model-design) evolution impact.

Table 1 Change impact metrics corresponding to a feature.

Metrics	Definition
NF	Counts the number features in a feature model
FNOP	Counts the number of packages in a feature
FNOC	Counts the number of classes in a feature
FNOM	Counts the number of methods in a feature
FNOA	Counts the number of attributes in a feature
FNOAs	Counts the number of associations in a feature

Table 2 Change impact metrics when adding a feature.

Metric	Definition
NF_added	Counts the number of added features
FNOP_added	Counts the number of packages added in a feature
FNOC_added	Counts the number of classes added in a feature
FNOM_added	Counts the number of methods added in a feature
FNOA_added	Counts the number of attributes added in a feature
FNOAs_added	Counts the number of associations added in a feature

Table 3 Change impact metrics when removing a feature.

Metrics	Definition
NF_removed	Counts the number of removed features
FNOP_removed	Counts the number of packages removed in a feature
FNOC_removed	Counts the number of classes removed in a feature
FNOM_removed	Counts the number of methods removed in a feature
FNOA_removed	Counts the number of attributes removed in a feature
FNOAs_removed	Counts the number of associations removed in a feature

3. Measuring the effort needed for evolutionary SPL change

To estimate the effort needed to implement an evolutionary change, several metrics have been defined in the software engineering field. The most well known metrics for object oriented applications were proposed by Chidamber and Kemerer (1994). By analogy, Lopez-Harrejon and Apel (2007) were interested in metrics for SPL. They define several metrics related to feature models like Number of Feature (NOF), Number Of Aspect (NOA), Number Of Classes and Interface (NCI), Base Code Fraction (BCF), aspect code fraction (ACF), introductions fraction (IF), advice fraction (AF).

In our work, we propose a set of new metrics adapted to SPL evolution that are inspired from Chidamber and Kemerer (1994). More specifically, we are interested in identifying the effort needed for change impact management in case of inter/intra feature model evolution. We propose a set of change impact metrics such as number of elements (features,

classes, attributes, methods, packages) added or deleted. Table 1 presents different metrics corresponding to a feature.

Table 2 presents different metrics measuring the effort needed when adding a feature and Table 3 when deleting it.

4. Feature model and design level change impact analysis

An SPL may evolve by the addition of some requirements, the modification of others, or by removing some requirements that are no longer useful. For example, today's mobile phones have a variety of common features, but manufacturers seek product differentiation by adding functions to attract consumers. This market evolution trigger has led to proposition of great innovations in the mobile phone industry. For example, with the evolution of mobile phones from their first generation 1G to their latest 4G, it is necessary to delete 1G's functional requirements and add 4G's requirements. In terms of SPL, the 1G related feature deletion induces the deletion of all its corresponding elements from the SPL design. Similarly, the addition of the 4G related feature must be aligned with the addition of its corresponding design elements (classes, methods, attributes) in an integrated way with the remaining design elements.

To ensure controlled evolutionary SPL changes, we propose a set of rules that define the pre and post-conditions for the change application and that identify the necessary changes at two levels: (1) intra-FM changes are changes relative to an artifact belonging to the solution space and corresponding to a feature (e.g., adding a class, a package or a set of classes and their code); and (2) inter-FM changes correspond to an evolution of a feature in the problem space (e.g., adding or deleting a feature).

4.1. Inter-feature model changes

The inter-FM changes concern the internal evolution of a feature model, such as adding, removing, and modifying a feature. We next present the rules to handle five different kinds of inter-FM evolution operations: feature addition, feature removing, feature renaming, feature moving and feature splitting.

4.1.1. Add feature

Suppose that the developer adds a feature to the feature model and would like to integrate the corresponding design fragment within the initial design. In this case, change impact management consists of adding the design elements corresponding to the feature while keeping the design consistent.

In order to add elements (classes, packages,...) corresponding to the new feature, we will use the following semantic criteria that express linguistic relationships between element names:

- Hypernyms(C1; C2,...,Cn): implies the name C1 is a generalization of the specific names C2,...,Cn, e.g., Media–Video.
- Synonyms(C1,...,Cn): implies that the names are either identical or synonym, e.g., Mobile–Mobile and Phone–Mobile.
- str_extension(C1; C2): implies that the name C1 is a string extension of the name of the class C2, e.g., Image-NameImage.

Table 4 Impact of adding a feature.

Change name	Add feature
Context	New requirement
Precondition	The new feature name is different from existing features names
Impact design	R1: If the name of a class A, belonging to the added feature, is a synonym of another class B, existing in the design, then the two classes will be merged R2: If a class A, belonging to the added feature, has a hypernyms or str_extension relation with another class B existing in the design, then B inherits A R3: If a class A, belonging to the added feature, has no relation with any other class B existing in the design, then the user has to choose classes and the relationship with the new added class

Table 4 presents the rules to apply when adding a new feature. These rules were inspired from works on model integration (Haddar et al., 2004).

For example, in Fig. 2, we propose to add the feature "Audio" which contains the class "PlayAudio", in this case we apply rule R2:

$$Hypernyms\ (PlayMedia, PlayAudio)$$
$$\Rightarrow \textit{The class PlayAudio'' inherits the class PlayMedia''}$$

Now, suppose that we want to add the feature "Mobile". This feature contains a class "Mobile" which contains an attribute "name" and a method "chooseMedia()". In this case, we apply R3 and we calculate the cosine similarity with the class having the highest similarity. In our context, we calculate the similarity between two vectors A and B by determining the angle between them. The vector A contains all terms that concern a class (class, methods and attributes names) and their synonyms. B contains all the features of the SPL with the names of their elements (see Fig. 3).

In our case, the vector A contains the elements (class: Mobile, attribute:name and method:chooseMedia()) and the vector B contains all the features (Media, Audio, Video) of the SPL with the names of their elements (class:PlayMedia, class:PlayVideo, class:AlbumPhoto and class:PlayAudio). In the running example, the value of the cosine similarity is 0.54, which means that the new class is similar to the class "PlayMedia". As a result, the new class will have a relationship with the class "PlayMedia".

4.1.2. Remove feature

Suppose that the developer deletes a feature from the feature model. In this case, change impact management consists in applying the rules in Table 5.

For the example of Fig. 4, let us remove the feature "Video" which has two descendants "Audio" and "Photo". "Video" is optional, as a consequence, we apply rule R4 and we calculate the degree of similarity between the features "Audio", "Photo" and the other features using the cosine similarity (Salton and Buckley, 1988). As a consequence, these two features will be moved to the feature "Media". By deleting the feature "Video", all elements belonging to this feature must be

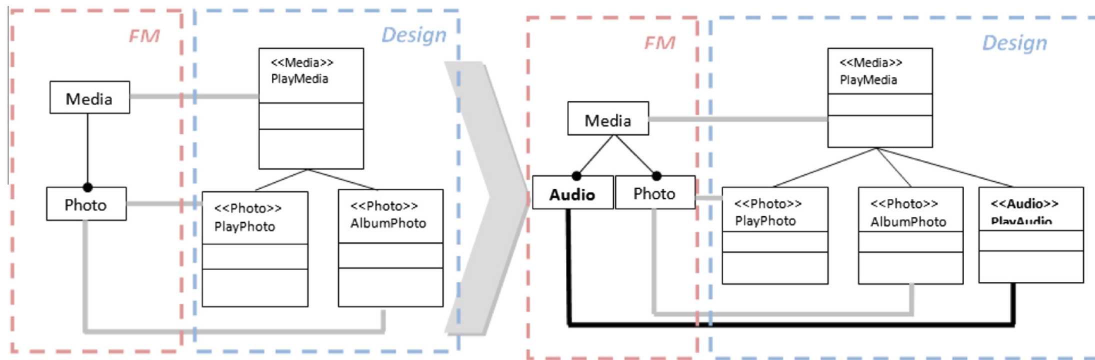


Figure 2 An example of a feature addition.

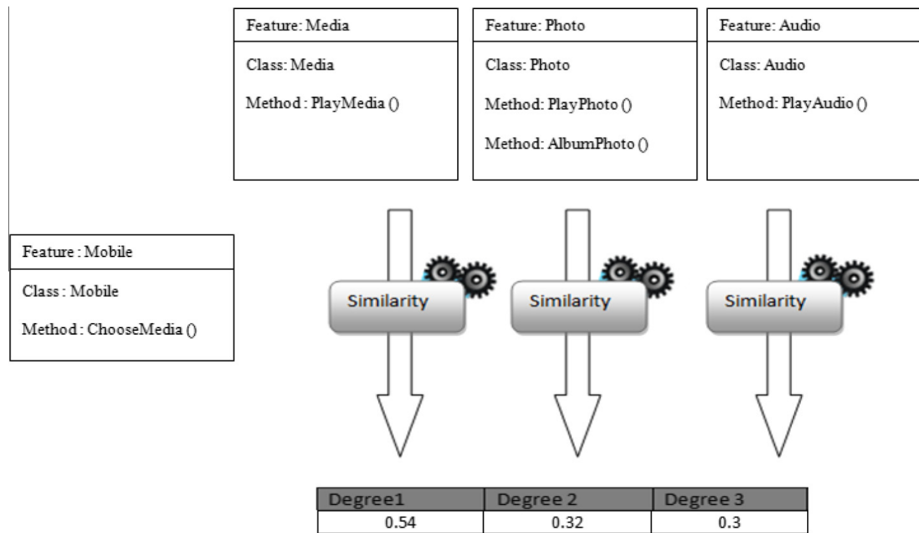


Figure 3 Similarity computation.

removed and as a result, the class "PlayVideo" must be removed according to the rules R5 and R6.

4.1.3. Rename feature

A consistent naming scheme improves product maintainability especially when feature names allude to the functions of the product. A feature may be renamed to reflect the underlying implementation changes, the adoption of different technologies, or the changes of application context. As a consequence, change impact management consists in replacing the stereotype "feature_name" in the design with the new name.

4.1.4. Move feature

The feature hierarchy may be reorganized. Moving a feature from a source composite feature to a target feature changes the parent-subfeature relation between the source feature and the target feature. By moving a feature, design will not be changed.

For example, in Fig. 5, we move the features "Audio" and "Photo". As a consequence, these features change their parent-subfeature, while, design does not change.

4.1.5. Split feature

A feature can be split into two or more sibling features. If it is a composite feature, some of its sub-features will be distributed (i.e., moved) to its new sibling features. Splitting a feature can be achieved by first adding new sibling features as leaf features and then moving some of the sub-features to the relevant new sibling features. Note that, when splitting a feature into two or more sibling features, the new features must have the same characteristics of the split feature (see Table 6).

For example, in Fig. 6, we split the feature "Media" in two features "AudioVisual" and "Visual". As a consequence, we apply the rule 10 and the new features are mandatory like the split feature.

4.2. Intra-feature model changes

The intra-FM changes concern the internal evolution of a feature. It corresponds to changes affecting the elements belonging to the feature (e.g., adding a class or a package or a set of classes and their code). In the following, three different kinds of intra-FM changes are presented:

Table 5 Impact of removing a feature.

Change name	Delete feature
Context	Obsolete feature
Condition	The deleted feature is not mandatory
Impact on the FM	R4: When deleting an optional feature, there are two possible cases: • If the feature has no descendants, then it will be removed • If the feature has descendants, then it is necessary to calculate the degree of similarity between the descendants and other features. According to this similarity degree, they are moved. If the cosine similarity value is under the threshold 0.7, then the descendants will not be deleted
Post condition	By deleting a feature, all elements of this feature must be removed. Elements can be packages, classes, methods or attributes. If a class is removed, all its relations (association, aggregation, composition) will be removed
Impact on design	R5: If an element (attribute, method, class, package) in the deleted feature F is used by another feature or associated by a conjunction of two features, then this feature will not be removed R6: If all the attributes and methods of a class are deleted, then this class will be also deleted

4.2.1. Add element

An SPL can be extended with new elements (package, class, method or attribute). The change impact rules when adding an element are presented in Table 7.

In order to add elements (classes, packages,...) corresponding to a feature, we will use R1, R2, R3 presented in Section 4.1.1.

For example, in Fig. 7, we add the class "PlayAudio", consequently for the impact on the FM, we apply rule R2:

Hypernyms (*PlayMedia*, *PlayAudio*)

⇒ *The class PlayAudio" inherits the class PlayMedia"*

On the other hand, the impact on the design consists in applying rule R11 and adding a new feature named "Audio".

4.2.2. Remove element

In some cases, there is a deletion of some parts of the design (packages, class, method and/or attribute). The rules of change impact management when removing an element are presented in Table 8.

In order to delete elements (packages, classes, methods...) from a design, we apply the rules R5, R6, already presented in Section 4.1.2.

For example, in Fig. 8, we propose to remove the class "PlayVideo". Note that the feature "Video" contains only this class. Thus, the change impact on the FM consists in applying rule R12 and as a consequence, the feature "Video" will be removed. On the other hand, the impact on the design consists in applying rule R5. Thus, when applying R5, "PlayAudio" and "PlayPhoto" will inherit from the class "PlayMedia" as illustrated in Fig. 8.

4.2.3. Rename element

When assigning new names to entities such as classes, methods or attributes, the change impact management consists in updating automatically all relevant occurrences of the names (references), method calls or references to attributes, in order to maintain the consistency of FM and design.

4.3. Impact classification

When a designer has a new requirement covering a (large) set of changes to the feature model, then the Change Advisory Board (CAB) produces a report containing the number of additional changes required to ensure the consistency of the design diagrams. The CAB can evaluate the changes to be made based on the impact it can have on the development process. In order to help CAB in decision making and to help him to decide on the changes to be made, an alternative consists in categorizing the impact. This classification of changes allows determining if the change is viable or not. The classification of impact implies the assignment of a category with regards

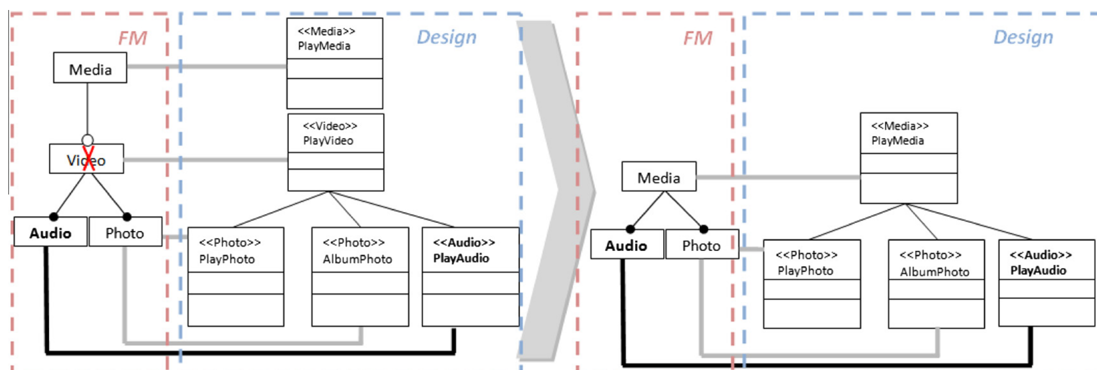


Figure 4 An example of a feature removal.

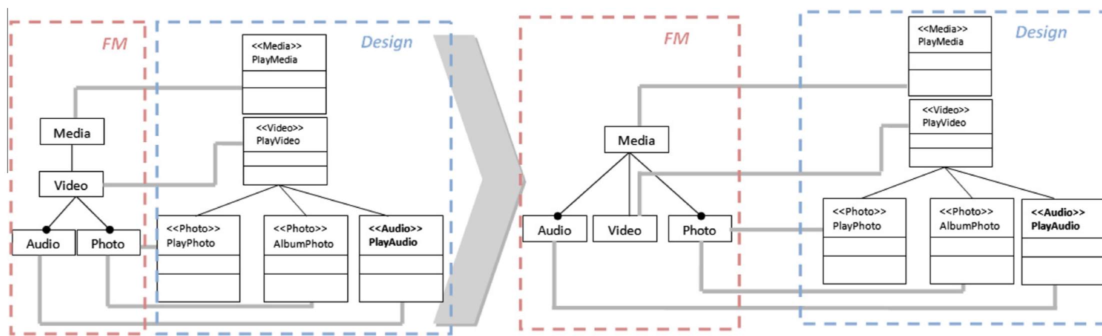


Figure 5 An example illustrating change impact when moving a feature.

Table 6 Impact of splitting a feature.

Change name	Split feature
Context	Requirement change
Condition	New features must have names that differ from existing feature names
Impact on the FM	R7: if a feature is optional, it will be split into new features that must be optional R8: if a feature is mandatory, it will be split into new features that must be mandatory R9: if a feature F1 has a relation "require" or "exclude" with another feature F2, all new split features must have the relation "require" or "exclude" with feature F2
Impact on design	R10: Every element (class, package, method, attribute) stereotyped with the name of the split feature F1, will be replaced with the name of the new features

to the decision-making authority. The category can be marginal, substantial or critical:

- Impact marginal: Change is related to a single product and side-effects can be safely excluded. Adding feature or its elements (class, method, attribute), splitting and renaming features are categorized as marginal impact.
- Impact substantial: Change affects several products of the SPL or it affects fundamental parts of the IT infrastructure, supporting several applications. It affects mandatory features or a feature that has an exclude/require constraint with another feature. Removing feature or elements of a feature are categorized as substantial impact.
- Impact critical: Change affects a major part of the business-critical infrastructure. It introduces major new technologies on a considerable scale.

5. Case study

To illustrate our approach, we use a Text Editing software product line as a case study. This family has eight product variants. Each product implements a simple Text Editing application. Features are collected in a FM to specify the variations between these products. The feature model of the Text Editing system is shown in Fig. 9. Note that the number of features is 30.

To help the user in managing change impact, while keeping a consistent SPL feature model and design, we developed a tool named "Evo-SPL". The main functionalities of Evo-SPL is the automatic calculus of the effort needed for change impact based on metrics and the generation of a new consistent SPL design and feature model after evolution. In fact, the user enters the feature model and the design before any evolution then he applies the changes on the feature model and on the design. Afterward, Evo-SPL verifies the rules and validates the feature model and the SPL design. The traceability between the feature model and design is presented with our UML profile. Finally, the effort needed for impact analysis is measured with our tool.

The source feature model is entered to the tool as an XML file. Then, he chooses the type of evolution. (i.e., evolution of the feature model or design). The user can add, split, remove or rename a feature. Moreover, he can add, remove or

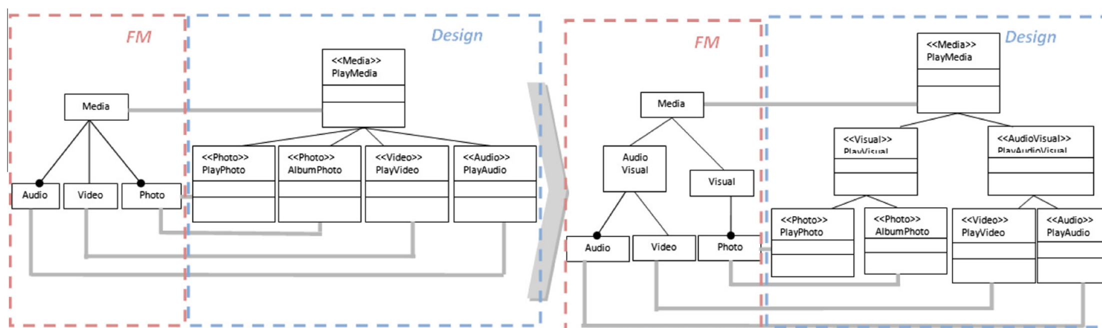


Figure 6 An example illustrating change impact when splitting a feature.

Table 7 Impact of adding an element.

Change name	Add element
Condition	The name of added element must be different from existing ones
Impact on the FM	R11: When adding a new element, there are two possible cases: • If the element has a semantic relationship with a feature F, then the element will be added to this feature F • If the element has no semantic relationship, then a new feature will be added to the feature model having the name of the added element
Impact on design	Rules R2, R3 presented in Section 4.1.1 are applied

rename an element which can be a package, class, method or attribute.

In our experiments, we applied 14 changes on the model. As an illustrative example, we next use the following change scenario to show how its change operations are handled through our approach and tool:

- Add a feature BASIC.
- Remove the feature SPLIT.
- Split the feature EDIT.

We measure the effort needed for impact management using the metrics presented in Section 3. Table 9 lists the metrics' values to measure the impact of the change scenarios.

We note that by adding two features, a class (FNOC_added), 4 methods (FNOM_added), 9 attributes (FNOA_added) and an association (FNOAn_added) will be added. By removing a feature, a package (FNOP_removed), a class (FNOC_removed), 2 methods (FNOM_removed), an attribute (FNOA_removed) and three associations (FNOAs_removed) will be deleted. Suppose that the user decides to perform the change scenario. In this case, the impact of change is presented as follows:

Change 1: Add a feature BASIC

Change impact: The new feature BASIC (illustrated in Fig. 10) has a name that is different from existing features names thus the precondition is satisfied. The feature is accompanied by a class named also "Basic" and containing two methods named "getBasic()" and "setBasic". No semantic relation (synonym, hypernyms, str_extension) is detected. Thus, rule R3 is applied. The highest value of the similarity calculus between the class "Basic" and the other classes (name of classes, methods, attributes) is found with the class "Text" and it equals 0.77. Then, our tool proposes to the user to choose a relation (association, aggregation and composition) between the new class "Basic" and the class "Text". In our case he chooses an association relation.

Change 2: Remove feature split.

Change impact: We note that the optional feature "Split" has descendants, thus, we apply the rule R4 and we calculate the degree of similarity between the descendants and other existing features. According to this similarity degree, we note that the cosine similarity between "splitVertical" and "changeDisplaySettings" is the highest value and it equals 0.744. Then, "splitVertical" is moved under "changeDisplaySetting" feature. Similarly, the cosine similarity is calculated for "splitHorizontal" and "Unsplit". As a consequence, the features "splitHorizontal" and "Unsplit" are moved under "changeDisplaySettings" feature (see Fig. 11). Moreover, all the elements in the class diagram that have the stereotype "Split" are removed.

Change 3: Split the feature EDIT.

The feature "EDIT" is split into two features named "Selection" and "Delete". By applying rule R8, these new features are stereotyped mandatory (see Fig. 12).

6. Evaluation

The overall objective of this section is to show the ability of our method and tool to detect the impacted elements and feature model after the SPL evolution and to produce an evolved feature model with a quality that is near to the quality of the initial feature model. For this purpose, we evaluated our method through a quantitative, empirical evaluation based on a comparison between feature models before and after evolution.

Besides the comparative evaluation, we conducted an expertise-based evaluation. It is based on a comparison between feature models where the change impact was obtained by applying our method and feature models where the impact was handled by experts. More specifically, we presented a list of changes to four experts who were asked to return the impacted features and elements in every feature as well as the corrected feature models. The participating experts are UML professionals and have previously studied and participated in SPL development projects.

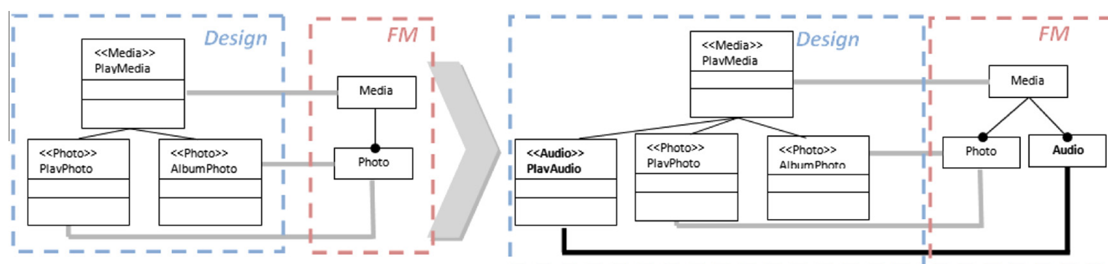


Figure 7 An example of adding a class.

Table 8 Impact of deleting an element.

Change name	Remove element
Context	
Condition	If the element is mandatory then it will not be deleted
Impact on FM	R12: If there exists a feature such that all its associated elements (classes, methods, attributes, packages) are removed, then the feature will be automatically removed

Table 9 Measuring the effort needed to manage the impact of the change scenarios.

Metrics	Value	Metrics	Value
NF	31		
NF_added	2	NF_removed	1
FNOP_added	0	FNOP_removed	1
FNOC_added	1	FNOC_removed	1
FNOM_added	4	FNOM_removed	2
FNOA_added	9	FNOA_removed	1
FNOAs_added	1	FNOAs_removed	3

6.1. SPL quality evaluation after evolution

Our empirical study took the following five feature models from the literature:

- FM1: Feature model for TankWar game.
- FM2: Feature model for MobileMedia system.
- FM3: Feature model for BerkeleyDB system.
- FM4: Feature model for Text Editing system.
- FM5: Feature model for Acrade Game Maker.

To compare the performance of our feature models after evolution, we used the metrics originally proposed in Dubslaff et al. (2014) and from which we took the following list:

- Number of features (NF): Counts the number of features in a feature model.
- Number of top features (NTop): Counts the number of features that are first direct descendants of the feature model root.
- Number of leaf features (NLeaf): Counts the number of features with no children or further specializations.
- Cyclomatic complexity (CC): Counts the number of distinct cycles that can be found in the feature model. Since feature models are in the form of trees, no cycles can exist in a feature model; however, integrity constraints between features can cause cycles. This metric counts the number of "exclude" and "require".

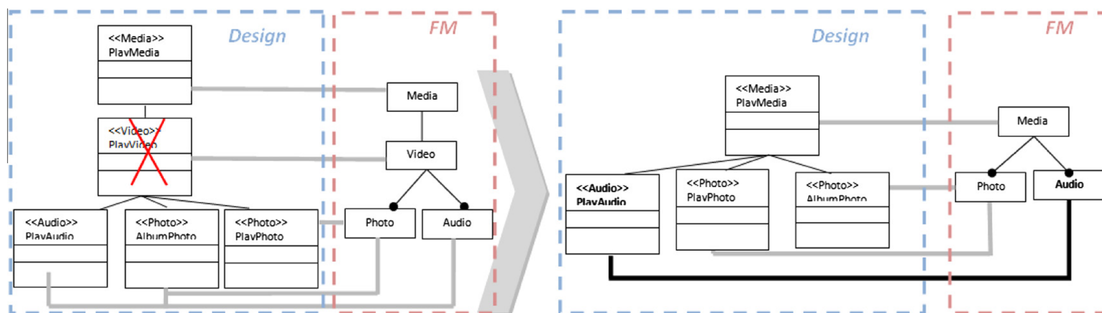


Figure 8 An example illustrating change impact when removing a class.

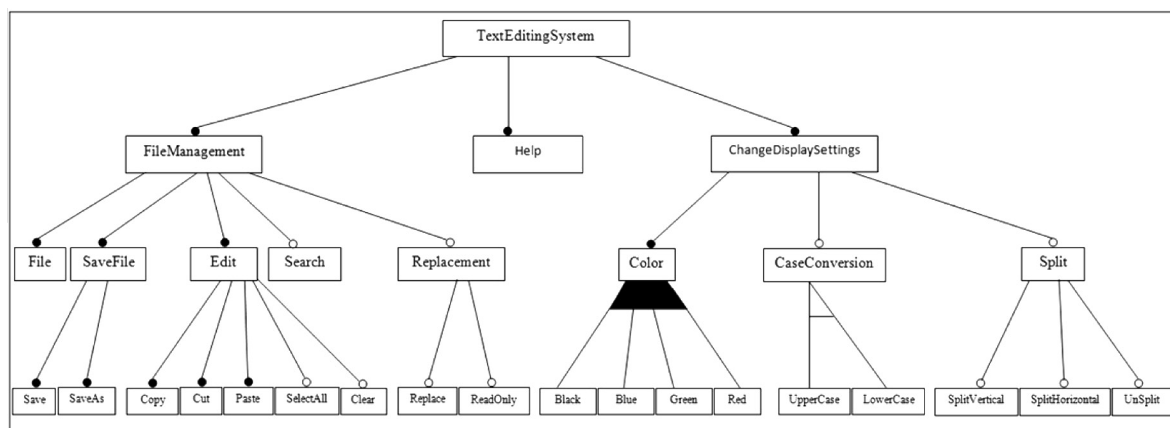


Figure 9 An example of FM of Text Editing.

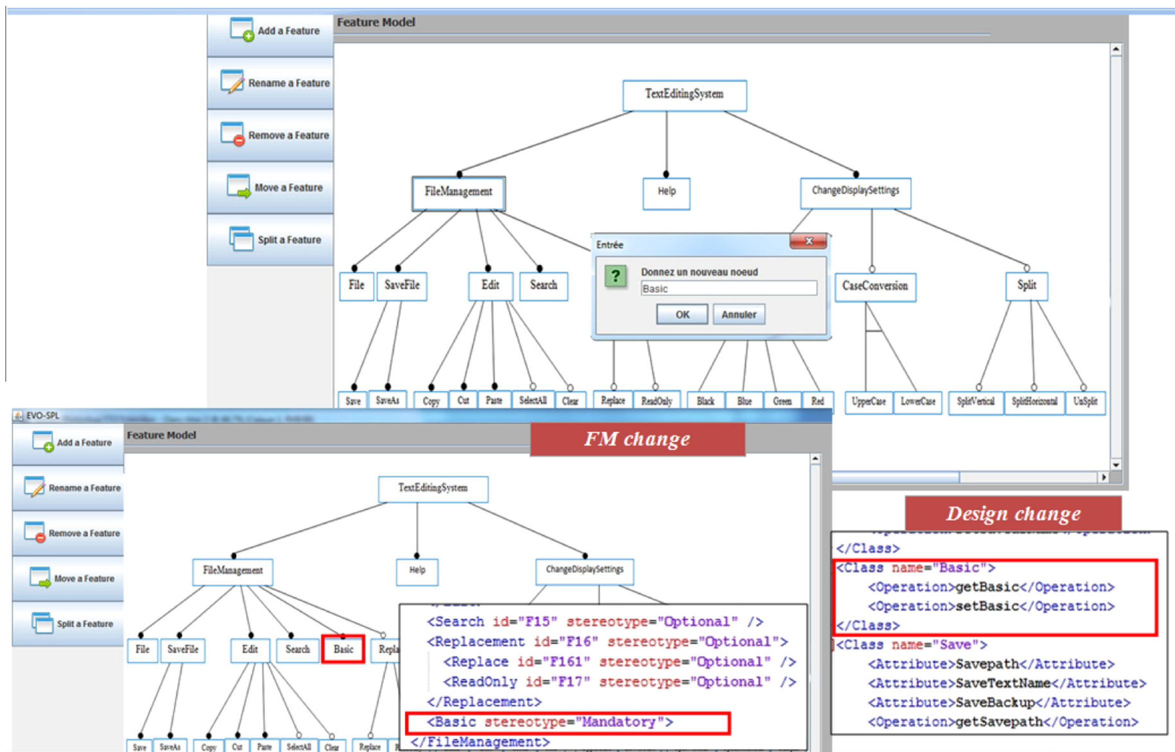


Figure 10 An example of adding a feature.

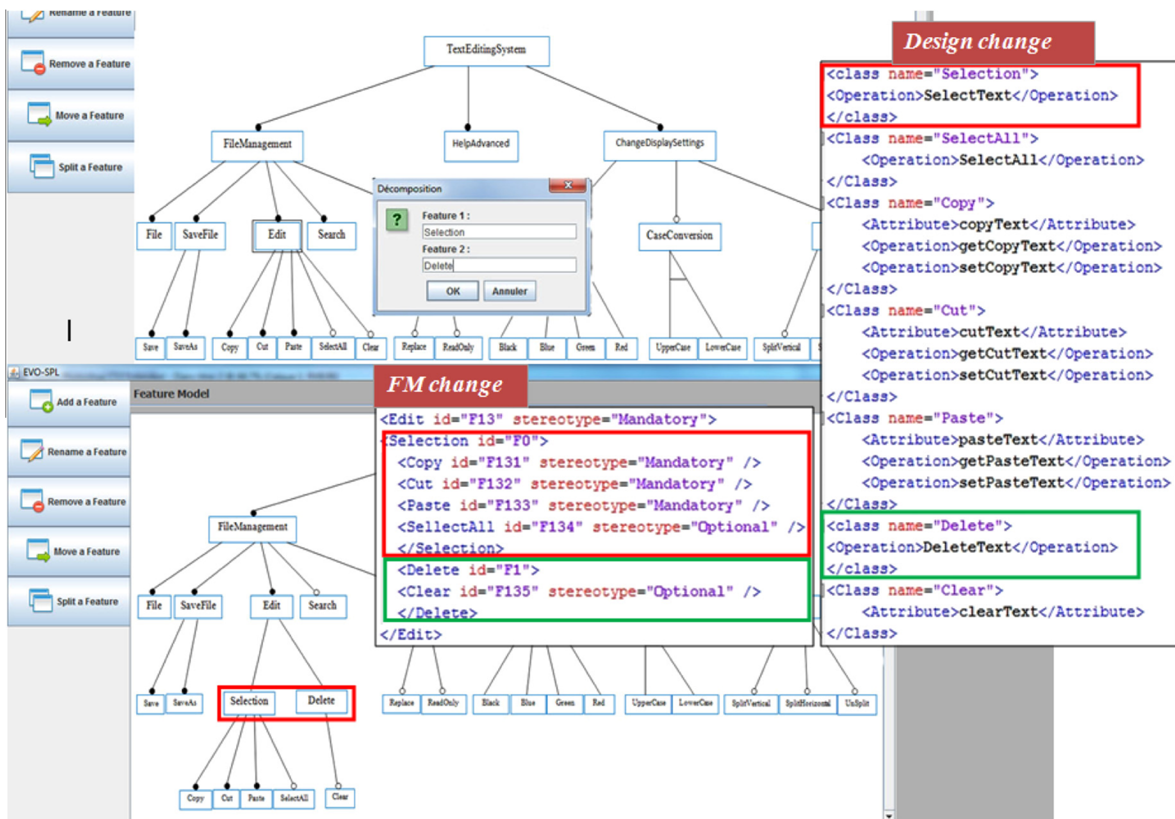


Figure 11 An example of removing a feature.

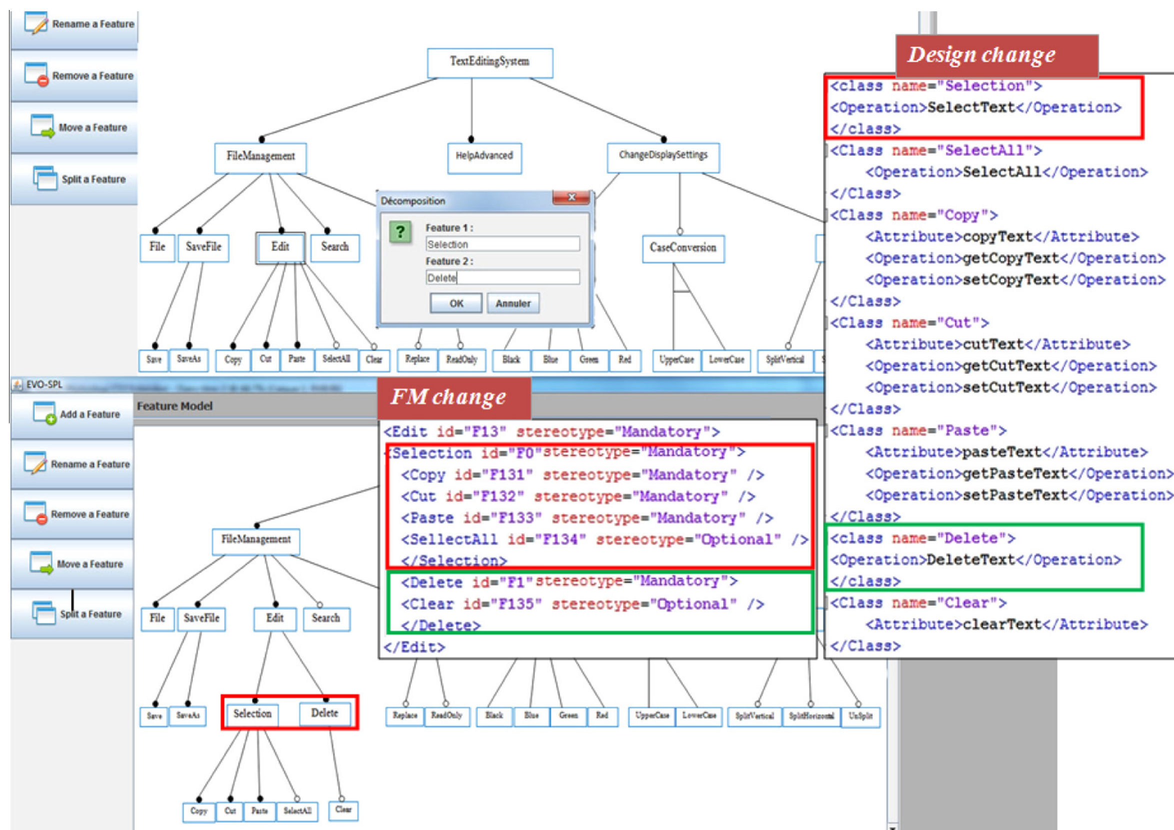


Figure 12 An example of splitting a feature.

- Ratio of variability (RoV): Counts the ratio of the average branching factor of the parent features in the feature model. In other words, the average number of children of the nodes in the feature model tree.
- Flexibility of configuration (FoC): Counts the ratio of the number of optional features over all of the available features in the feature model.
- Coefficient of connectivity density (CoC): Counts the ratio of the number of edges over the number of features in a feature model.

Fig. 13 shows a comparison of quality metrics values obtained for the feature model of TankWar game, MobileMedia system, BerkeleyDB system, Text Editing system and Acrade Game Maker before applying any change and the feature model obtained by our approach and tool to take into account the changes. For the first feature model, four features are added and one deleted. For the second, two features are added. For the third, three features are added and two are deleted. For the fourth, one feature is added and finally, for the fifth one feature is added and three are deleted. It is clear that the values obtained by our approach are close to those obtained for the feature model resulting from the work of experts and without any change; this is indicated clearly in the different curves shown in Fig. 13.

In conclusion, our preliminary empirical study shows that the feature models generated after evolution are of high quality because they do not go beyond the values of the used metrics applied on other feature models.

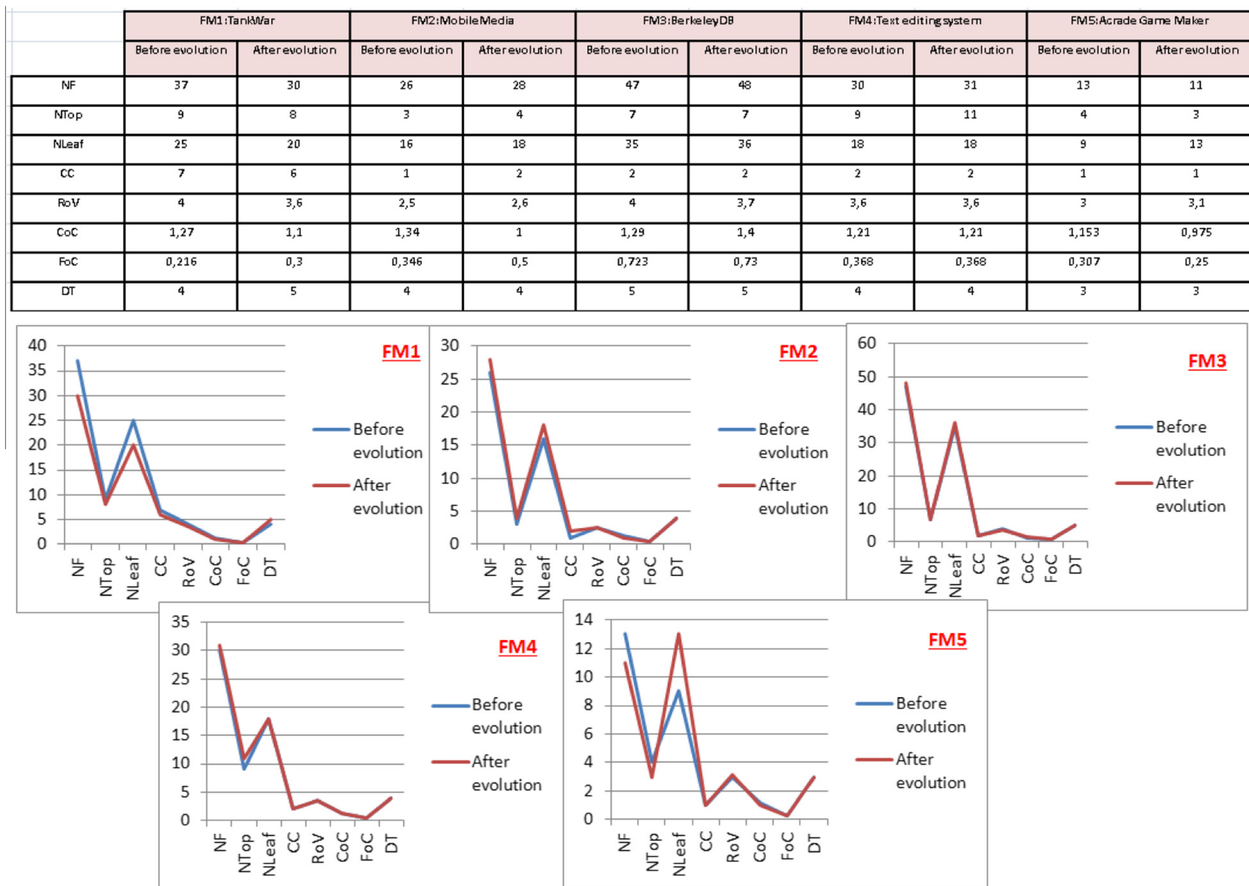
6.2. Change impact evaluation

For evaluation purposes, we also used the recall and precision measures: precision represents the number of correct impacted changes detected by our tool among all the impacted elements found by our tool. Recall represents the number of correct impacted elements belonging to the feature detected by our tool among all the existing real impacted elements belonging to the feature. Moreover, we count the number of True Positives (TP), False Positives (FP), and False Negatives (FN). False positives are impacted elements belonging to the feature wrongly identified. False negatives are actual impacted elements belonging to the feature that have not been detected by our approach.

$$\text{Precision} = \frac{\text{Number of correct impacted features detected by our tool}}{\text{Number of found impacted featured found by our tool}}$$

$$\text{Recall} = \frac{\text{Number of correct impacted features detected by our tool}}{\text{Number of existing real impacted features}}$$

In our evaluation (Table 10), the average precision of 0.79, is explained by the fact that we found some false positive impacted features (i.e., incorrect detected impacted features). Compared to the true positives found by our method, the false positives impacted elements are not significant. The recall, whose average value is 0.95, indicates that we have also some false negative impacted features (i.e., true impacted features not detected).



	FM1:TankWar		FM2:MobileMedia		FM3:BerkeleyDB		FM4:Text editing system		FM5:Acrade Game Maker	
	Before evolution	After evolution	Before evolution	After evolution	Before evolution	After evolution	Before evolution	After evolution	Before evolution	After evolution
NF	37	30	26	28	47	48	30	31	13	11
NTop	9	8	3	4	7	7	9	11	4	3
NLeaf	25	20	16	18	35	36	18	18	9	13
CC	7	6	1	2	2	2	2	2	1	1
RoV	4	3,6	2,5	2,6	4	3,7	3,6	3,6	3	3,1
CoC	1,27	1,1	1,34	1	1,29	1,4	1,21	1,21	1,153	0,975
FoC	0,216	0,3	0,346	0,5	0,723	0,73	0,368	0,368	0,307	0,25
DT	4	5	4	4	5	5	4	4	3	3

Figure 13 A comparative study by measurement.

Table 10 Evaluation results.

Evaluation	TP	FP	FN	Precision = TP/(TP + FP)	Recall = TP/(TP + FN)
Comparative	46	16	3	0.74	0.93
Expertise	39	7	2	0.84	0.95
Average				0.79	0.94

The sum of the true positives and the false negatives is equal to the total number of actual change impacts. These false negatives can be explained by the fact that similarity calculus which permits to calculate the similarity between features names, classes names, attributes names, gives similar values in some cases. In this case, our tool can not define the true change.

For example, if we want to remove a feature "Sound" which has descendants, we apply the rule R4 presented in Section 4.2.2 and we calculate cosine similarity between the descendants and other existing features. When we calculate the similarity between the descendant feature "Play" and features "video", "audio", we found that values are identical and equals to 0.7. In the case which we found more than one identical value, our tool does not found under which feature the feature "Play" will be moved.

7. Conclusion

This paper presented a new method for SPL change impact management. Unlike existing methods for SPL change management which operate only on the feature model, the proposed automated method helps the designer to preserve the consistency of the design and the feature model. It identifies the set of changes required to propagate each type of change from the feature model to the design. The propagation is ensured thanks to our UML profile for SPLs, which explicitly links the SPL feature model to its design.

Our future works concern another evaluation where we compare the feature model and design after an evaluation treated by experts and the feature model and design after the same evolution and obtained by our approach rules. We will also be interested in the formalization of our change impact rules and

we will be interested in the configuration management of software product lines.

References

Acher, M., Baudry, B., Heymans, P., Cleve, A., Hainaut, J., 2013. Support for reverse engineering and maintaining feature models. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS'13, New York, NY, USA, pp. 1–8.

Al-Msie'Deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H., 2012. An approach to recover feature models from object-oriented source code. In: Day Product Line 2012.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. In: Alves, V. (Ed.), Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE'06, New York, NY, pp. 201–210.

Barry, W.B., 1991. Software risk management: principles and practices. IEEE Softw. 8 (1), 32–41.

Ben-Abdallah, H., Bouassida, N., Gargouri, F., Hamadou, A.B., 2004. A UML based framework design method. J. Object Technol., 97–120

Borba, P., Teixeira, L., Gheyi, R., 2010. A theory of software product line refinement. In: Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing, ICTAC'10. Springer-Verlag, pp. 15–43.

Botterweck, G., Pleuss, A., 2014. Evolution of software product lines. In: Evolving Software Systems, pp. 265–295.

Chidamber, S., Kemerer, C., 1994. A metric suite for object oriented design. IEEE Trans. Software Eng., 476–493

Clements, P., Northrop, L., 2001. Software product lines: practices and patterns. SEI Series in Software Engineering.

Cordy, M., Classen, A., Schobbens, P., Heymans, P., Legay, A., 2012. Managing evolution in software product lines: a model-checking perspective. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS'12, pp. 183–191.

Czarnecki, K., Eisenecker, U., 2000a. Generative Programming – Methods, Tools and Applications. Addison-Wesley.

Czarnecki, K., Eisenecker, U., 2000b. Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA.

Dubslaff, C., Klüppelholz, S., Baier, C., 2014. Probabilistic model checking for energy analysis in software product lines. In: Proceedings of the 13th International Conference on Modularity, MODULARITY'14, pp. 169–180.

Guo, J., Wang, Y., 2010. Towards consistent evolution of feature models. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, pp. 451–455.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., 2012. Evolving delta-oriented software product line architectures. In: Monterey Workshop, vol. 7539, pp. 183–208.

Haddar, N., Gargouri, F., BenHamadou, A., 2004. Une approche formelle pour l'intégration des aspects structuraux et comportementaux de représentations conceptuelles. In: Maghrebian Conference on Software Engineering and Artificial Intelligence.

Heider, W., Rabiser, R., Grünbacher, P., Lettner, D., 2012. Using regression testing to analyze the impact of changes to variability models on products. In: Proceedings of the 16th International Software Product Line Conference – Volume 1, SPLC'12, pp. 196–205.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., 1990. Feature-oriented domain analysis (foda) feasibility study, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

Laguna, M., Crespo, Y., 2013. A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. Sci. Comput. Program 78, 1010–1034.

Loesch, F., Ploedereder, E., 2007. Restructuring variability in software product lines using concept analysis of product configurations. 159–170.

Lopez-Harrejon, R., Apel, S., 2007. Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies. In: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, FASE'07, pp. 423–437.

Lozano, A., 2011. An overview of techniques for detecting software variability concepts in source code. In: ER Workshops, pp. 141–150.

Maazoun, J., Bouassida, N., Ben-Abdallah, H., 2013. Feature model extraction from product source codes based on the semantic aspect, ICSOFT13.

Maazoun, J., Bouassida, N., Ben-Abdallah, H., 2014. A bottom up SPL design method, MODELSWARD'14.

Mende, T., Beckwermert, F., Koschke, R., Meier, G., 2008. Supporting the grow-and-prune model in software product lines evolution using clone detection. In: CSMR. IEEE, pp. 163–172.

Michalik, B., Weyns, D., 2011. Towards a Solution for Change Impact Analysis of Software Product Line Products. IEEE Computer Society, pp. 290–293.

Neves, L., Teixeira, L., Sena, D., Alves, V., Kulezsa, U., Borba, P., 2011. Investigating the safe evolution of software product lines. SIGPLAN Not. 47, 33–42.

Paskevicius, P., Damasevicius, R., tuikys, V., 2012. Quality-oriented product line modeling using feature diagrams and preference logic. In: Information and Software Technologies. 241–254.

Passos, L., Guo, J., Leopoldo, T., Czarnecki, K., Wasowski, A., Paulo, B., 2013. Coevolution of variability models and related artifacts: a case study from the linux kernel. In: 17th International Software Product Line Conference. ACM.

Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S., 2012. Model-driven support for product line evolution on feature level. J. Syst. Softw. 85 (10), 2261–2274.

Ramil, F.J., Lehman, M.M., 2000. Metrics of software evolution as effort predictors – a case study. In: Proceedings of the 16th International Conference on Software Maintenance. IEEE Computer Society, pp. 163–172.

Romero, D., Urli, S., Quinton, C., Blay-Fornarino, M., Collet, P., Duchien, L., Mosser, S., 2013. Splemma: a generic framework for controlled-evolution of software product lines. In: SPLC Workshops. ACM, pp. 59–66.

Salton, G., Buckley, C., 1988. Term-weighting approaches in automatic text retrieval. In: Information Processing and Management. 513–523.

Schaefer, I., 2010. Variability modelling for model-driven development of software product lines. In: VaMoS, vol. 37. pp. 85–92.

Schubanz, M., Pleuss, A., Pradhan, L., Botterweck, G., Thurimella, A., 2013. Model-driven planning and monitoring of long-term software product line evolution. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS'13, New York, NY, USA. pp. 18:1–18:5.

Schulze, S., Thüm, T., Kuhlemann, M., Saake, G., 2012. Variant-preserving refactoring in feature-oriented software product lines. In: VaMoS, pp. 73–81.

Seidl, C., Heidenreich, F., Amann, U., 2012. Co-evolution of models and feature mapping in software product lines. In: SPLC (1). pp. 76–85.

Shahzad, A.A.-M.B., 2010. Risk identification and preemptive scheduling in software development life cycle. Global J. Comput. Sci. Technol. 10, 55–63.

Shahzad, B., Iqbal, J., 2007. Software risk management prioritization of frequently occurring risk in software development phases using relative impact risk model. In: Proceedings 2nd International

Conference on Information and Communication Technology, pp. 110–115.

Shen, L., Peng, X., Zhao, W., 2009. A comprehensive feature-oriented traceability model for software product line development. In: Australian Software Engineering Conference, pp. 210–219.

Thüm, T., Batory, D., Kastner, C., 2009. Reasoning about edits to feature models. In: Proceedings of the 31st International Conference on Software Engineering, ICSE'09. IEEE Computer Society, Washington, DC, USA, pp. 254–264.

Xue, Y., 2011. Reengineering legacy software products into software product line based on automatic variability analysis. 1114–1117.

Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M., 2013. Feature identification from the source code of product variants. 417–422.