Using Composite Feature Models to Support Agile Software Product Line Evolution

Simon Urli
UNS, CNRS (I3S - UMR 7271)
Nice, France
urli@i3s.unice.fr

Mireille Blay-Fornarino
UNS, CNRS (I3S - UMR 7271)
Nice, France
blay@i3s.unice.fr

Philippe Collet
UNS, CNRS (I3S - UMR 7271)
Nice, France
collet@i3s.unice.fr

Sébastien Mosser
SINTEF ICT
Oslo, Norway
sebastien.mosser@sintef.no

ABSTRACT

Managing continuous change in a *Software Product Line* (SPL) is one of the challenges now faced by the SPL engineering community. On the one hand, the SPL paradigm captures the intrinsic variability of a software based on a systemic vision of the software to model. On the other hand, Agile Software Development advocates the incremental development of software based on constant interaction with a customer community. In this paper, we present an approach based on *Composite Feature Models* (CFM) to support the agile evolution of a SPL. This study is driven by the refactoring of a daily used application (information broadcasting system), in the context of a nationally funded project. Preliminary results show that CFMs support the incremental development of a SPL based on interactions with a community, tackling the challenge of SPL continuous evolution.

Keywords

Software product line, Agile, Evolution, Feature model

1. INTRODUCTION

In many domains software maintenance and evolution lead to abstract the domain concepts in a model-driven engineering process while supporting variability and extension points. The latter corresponds to the application of the *Software Product Line* (SPL) paradigm, which consists in systematically reusing development assets in an application domain [8], reducing time-to-market and improving quality.

Still willing to improve software development efficiency, *Agile Software Development* (ASD) advocates the incremental development of a software based on constant interaction with a customer community [12]. Contrary to SPL techniques, in ASD the focus is neither on scoping, predictability nor reuse. ASD primarily aims at addressing the immediate customer demands with development efforts at a smaller scale. While both approaches aim at handling variability, they are not built to be directly used together, even if some studies have been already conducted [7, 15, 13].

Nowadays software-intensive systems are built in a day-to-day manner through communities that follow the rules of an architectural framework, similar to a SPL. A crucial issue is then to evolve towards a community-driven SPL, following agile principles. The community would constantly provide new requirements and change requests, while the reasoning and generative capabilities provided by the SPL framework should be preserved or adapted accordingly, following agile principles. Based on our current experience on the reengineering as a SPL of a concrete and daily used application, an information broadcasting system, we propose in this paper an approach to support the agile evolution of an SPL.

After discussing the current state of the art on agile SPL evolution (see Section 2), we introduce our case study and discuss issues related to its reengineering as a SPL (see Section 2). We show that current variability modeling techniques are not powerful enough to handle the faced problems. We discuss how the evolution of domain objects can be managed by introducing an extension of feature models [16], (a widely used formalism for variability modeling), called *Composite Feature Models* (CFM, see Section 4). To manage dependencies between these evolving objects, we describe a model-driven configuration process (see Section 5). We then conclude by discussing remaining difficulties in our proposal and by evoking future work (see Section 6).

2. ON AGILE SPL EVOLUTION

The SPL paradigm models the intrinsic variability of a family of software systems based on a systemic vision of the software to model. It usually relies on reusable artifacts that encapsulate both common and variable aspects so as to facilitate planned and systematic reuse [8]. An SPL is by nature scoped, *i.e.*, it handles changes inside the software family by configuration, providing a tailored product. This means that all the changes are known in advance and captured in some variability models, such as *Feature Models* (FM) [16]. Evolving an SPL out of its original scope is therefore a challenging activity, as it breaks the closed world assumption.

Some recent work focuses on providing safe evolution principles for SPLs [19], but are restricted to change patterns

ME '12, September 30 2012, Innsbruck, Austria

that follow a refinement notion. At the FM level, we observe the application of classic software engineering approaches: reasoning on FM edits [20], providing some consistency notions when FM evolves [14] or semantically differentiating FMs [3]. But in many domains now, several factors raise the stakes on the SPL engineering techniques and call for appropriate evolution management techniques: third party components are used, software ecosystems are moving to open-source and community-based management, thus continuous requirement changes must be handled [5]. In this context, managing continuous change in an SPL becomes a crucial challenge, to which agility provides answers [12].

The interactions between SPL engineering and agility has been studied according to several viewpoints. The usage of agile techniques when tailoring a product within the fixed scope of a SPL was investigated in [7]. Different industrial studies show some complementarity in the planning, with long-term strategy devoted to SPL management and short-term tactical decisions with agile principles [15], as well as some need to change team organisation according to the compositional form of today SPLs [5]. Besides a test-driven approach is also proposed in [13], where the variability is incrementally introduced into software by refactoring existing code in a agile way.

To the best of our knowledge, no approach has been provided to handle the evolution of a SPL (especially community-driven), following agile principles. We advocate that this problem can be tackled by managing the evolution of several notions and software artifacts. First, as the domain is constantly evolving, the evolution management of domain objects must be handled, taking into account the complexity of both the objects and their variability models. Second, dependencies between these domain objects must be handled, facilitating both the direct usage of simple users of the SPL, and the long-term management of the SPL by their architects.

3. FROM JSEDUITE TO YOURCAST

This section presents our case study, the re-engineering of a broadcasting system as a SPL.

3.1 JSeduite : the legacy system

The development of the jSeduite[1] system started in 2004. Used to broadcast information in academic institutions, the system is connected to academic partners (*e.g.*, transportation network, cafeterias) that act as *sources* of information. This information is then dispatched to multiple devices (*e.g.*, public screens, smartphones) that *render* it. The system is implemented as a *Service-Oriented Architecture* (SOA), and has been recognized as a prototypical usage of SOA [18], used as a validation case study in the Faros national project. The final version of jSeduite was released in 2010, and counts circa 70,000 lines of code. This system is today used in several french campuses.

Two kinds of persons interact with jSeduite. On the one hand, final *users* (*e.g.*, students, lecturers) see the broadcasting devices. They express their needs in terms of information delivery (*i.e.*, a "product configuration" according to the SPL terminology) using their own vocabulary, *e.g.*, "I want to display the timetable and the news from the University on cafeteria's screens". On the other hand, *developers* work at the code level to implement web services (*e.g.*, timetable provider, news feed reader) and business processes orchestrating these services to support information broadcasting (*e.g.*, aggregating the two services to merge their information streams). Developers also interact with the code of the display client, customizing it to their institution (*e.g.*, logo, color codes, layout).

One of the interesting dimension of jSeduite is that the system federates a community of users and developers. The system was started as an engineering school project, and was quickly enriched through student initiatives, adding new sources of information to the system. Several *broadcasting policies* were empirically identified to manage the growing set of available information on the screens (*e.g.*, restricting to the latest ones, prioritizing certain kinds of information during breaks). Two institutions dedicated to visually impaired children are involved in this community, which triggers new challenges in the way information is rendered through the GUI (*e.g.*, text-to-speech, dedicated fonts and color codes).

3.2 YourCast: toward an agile SPL

The jSeduite system suffered from several issues. First, it was originally targeting small institutions. Thus, we encountered scalability problems while deploying a jSeduite instance on large institutions (*e.g.*, multiplicity of users, variability of devices). The second issue is triggered by the richness of the source set and their associated parameters, which make it difficult to customize as is. Empirically, we identified that the customization of a jSeduite instance by an end user is tedious and error-prone.

The YourCast project[2] aims to address these two issues, using a SPL approach to support end-users. An end-user wishing to have a broadcasting system selects the wanted features, and the YourCast engine generates the final product based on this selection. This is directly related to the first agile principle: "*satisfy the customer*" [12, principle 1]. Moreover this SPL should evolve to include the contributions of a community of developers, creating new pieces of software, or modifying them to enrich and maintain the product family. We can make a parallel here with two other agile principles: the need to "*embrace change*" and to "*give frequent deliveries*" [12, principles 2 and 3]. Each contribution of the community to evolve the SPL comes from a change and must give at least a new usable delivery of the system.

We distinguish two groups of stakeholders: *(i)* users of the SPL itself that create a *Broadcasting System* (BS) by selection and then generate it. We call them BS-User; *(ii)* developers who enrich the SPL. They form a community and we note them BS-Dev.

From the perspective of a BS-User, we identified the following concepts: a BS is composed of a set of *sources*, each one being formatted to their liking to be displayed (*renderer*) in a *layout*. Before it can be processed by one *policy* or more to filter or sort pieces of information. We represent in Fig. 1 a simplified version of the domain model. Each class of the model represents the domain concepts. The arrows and the cardinalities in bold represent the requirements to make a valid broadcasting system (*e.g.*, *"at least one source"*). The other lines and cardinalities represent dependencies between concepts (*e.g.*, *"a source need to be connected with exactly one renderer"*).

[1] `http://www.jseduite.org`

[2] `http://www.yourcast.fr`, ANR EMERGENCE

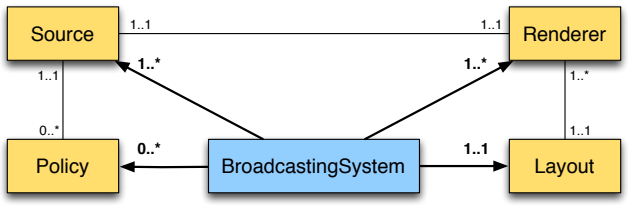

Figure 1: Simplified domain model of BS

The purpose of a BS-User is to create a BS that is aligned with her information system and the sources she finds on the web, while benefiting from a display in adequacy with her context of use, *e.g.*, graphic standard, type of population (*e.g.*, young, disabled, corporate), or field conditions. Goals of developers (BS-Dev) are multiple, *e.g.*, to add or adapt sources, create its own renderers, offer new layouts. These developments are driven by the consideration of technological changes and by needs from the field experiments.

4. EVOLUTION OF DOMAIN OBJECTS

The evolution of the Web 2.0 increases the possible sources of information (*e.g.*, *RSS feeds, user-driven news channels*) by the publication of shared or legacy services. This evolution translates into a reduced time-to-market for new business solutions. The YourCast SPL must in turn support this evolution by demonstrating its ability to absorb evolving requirements and priorities.

4.1 Requirements

R1: Reactivity to domain evolution. Broadcasting systems embrace different sub-domains (*e.g.*, sources, visualization technique, display, policies) that are in constant evolution. Our developer community aims to integrate these new domain objects in existing or new broadcasting systems. Yourcast must support the integration of these various domain objects. For example, if a new service to share pictures is widely used, a BS-Dev might decide to implement software assets to use it in a broadcasting system. These assets have to be reified in the SPL.

R2: Reactivity to domain objects evolution. Adding new domain object is not sufficient to support the evolution of the SPL. Services on the web are constantly evolving: APIs are changing and new features appear. The BS-Dev have to modify the SPL, reifying the changes. For example, the previously defined service offers now the capacity to manage secured picture albums, and a BS-Dev adds this feature in the SPL.

R3: Human understability. For a simple broadcasting system, we need to use many concepts. Each concept has common and variant parts that represent the different domain objects available. But even the domain objects have their own variability that has to be represented. For example, a *source* like *Picasa* must be represented with all the relevant ways to use it: it is possible (the list is not exhaustive) to use the service as a search engine, or to get a precise picture album, to sort pictures by date or by alphabetic order. That means a simple domain object could be represented by a large FM reflecting what the BS-User wants to manipulate (*e.g.*, the Picasa source corresponds to 18 features in the running system). In the end, getting the variability of all the domain objects quickly leads us to a very complex FM. So BS-Dev need some ways to manage that complexity. In our example, 7 domain object *sources* correspond to 27 features and 36 valid configurations, 35 domain object *renderers* correspond to more than 50 features.

R4: User-driven vocabulary. The last need concerns the vocabulary employed in the SPL. Empirically, the introduction of objects in the SPL requires to characterize the domain objects in terms faimiar to the BS-User. This step is difficult. Therefore it is necessary to support vocabulary changes in the SPL itself following the experiments. For example, the *"screen_name"* term comes from Twitter API and is used to filter information (thus modeled as a feature). But it is not as clear as *"pseudonym"* to a user who does not know the Twitter API.

4.2 Solution: Composite Feature Models

This section describes the solution defined in the context of the YourCast national project to address these requirements. This solution is based on the definition of *Composite Feature Models*, according to the following pillars: *(i)* separation of concerns, *(ii)* bottom-up modeling and *(iii)* automated refactoring.

Separation of concerns. The usual way to represent a system using a FM consists in gathering information about the domain and the existing system and in creating a FM with this knowledge. In case of evolution, the designer of SPL knows all about the FM and computes the changes. Our first requirement (R1) triggers two challenges: first, we need to manage different domains for each concept used in the global SPL; second, in each of these domains new domain objects appear and should be modeled in the SPL. Moreover, the third requirement (R3) is directly linked to the *"keep things simple"* agile principle [12, principle 10]. Splitting a large problem into many small ones is a well known solution to manage complexity and is widely encouraged in software engineering generally through principles like separation of concerns.

SPLs do not intrinsically support separation of concerns. In front of the need to model a SPL using several models to be composed, the notion of *multiple SPL* was identified [10, 3]. According to this principle, we propose to create a FM for each domain concept of the SPL. The set of FMs forms the *Composite Feature Models* (CFM). In our case study, we have a FM to represent sources, renderers, policies and layouts. As a consequence, we manipulate smaller FMs in the order of few tens of features and each FM is dedicated to a precise domain and not to the large domain of broadcasting systems. The complete SPL is thus composed of a set of FMs, each one corresponding to a different domain concept.

Fig. 2 depicts two examples of partial FMs for Source and Renderer. We can see three domain objects of source representing the services *Picasa*, *FlickR* and *Twitter* in the FM *Source*; the FM *Renderer* shows three domain objects of renderer usable with *tweets* and *picture albums*: the two renderers for picture albums represented them in different ways, *MosaicAlbum* displays albums in a picture mosaic while *SlideAlbum* displays slides of pictures. As a BS-User,

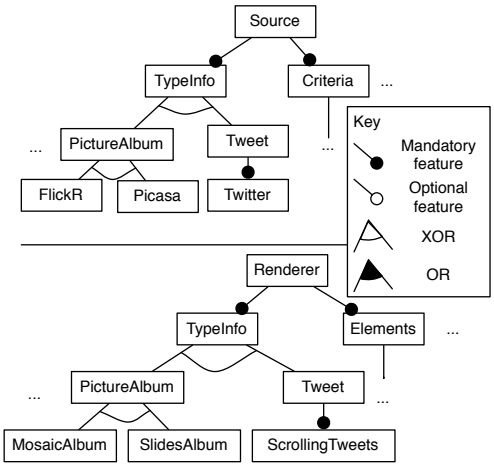

Figure 2: Partial FMs for Source and Renderer

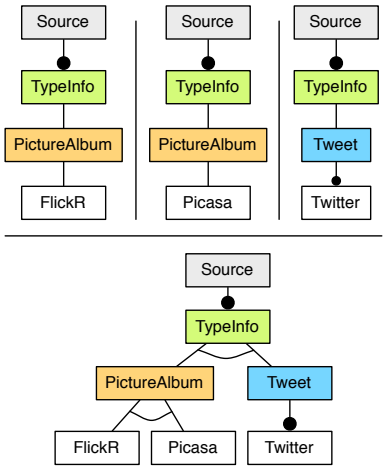

Figure 3: Using a merge algorithm to build a FM

it is now possible to focus on these domain concepts independently. While configuring a system, a BS-User interested in rendering mechanisms opens the *Renderer* FM, and can select the expected features without being overwhelmed by the other domain concepts (*e.g.*, sources of information).

Bottom-up modeling. Separating general and domain object specific functionalities is non trivial *a priori* [6]. Even with smaller FMs evolutions are tedious and error prone. Modifying a FM to reflect a change in a domain object as it is specified in the second requirement (R2) needs to perfectly understand the whole FM. But it is contradictory with the idea of a community making evolutions: in that kind of process, each member of the community has a little piece of knowledge. Moreover, the third requirement (R3) emphasizes the need for complexity management.

To address this point, we assume that each element to be added or modified exists as a software artifact. Thus, it is possible to model this artifact and the associated ones as a FM, based on the local knowledge. The integration of this "partial" FM dedicated to a given evolution into the composite feature model is delegated to an automated algorithm used to merge them together [2]. The algorithm can be used in union, intersection or diff mode to compute a feature model supporting respectively: the union of configurations providing by feature models in input ; the intersection of configurations existing in feature models in input ; and the difference of possible configurations. We use it here in union to propagate the evolution of the domain objects into the existing FM, freeing the BS-Dev from this tedious task. The operation will automatically add the new available configurations to the old ones.

So the creation of FMs which are used in SPL is an incremental and automated process: if a change in a domain object has to be reflected in FMs, the BS-Dev only makes modifications in the domain object FM and then do a merge of the set of FMs. If a domain object is deleted, the BS-Dev just has to delete the associated feature model and to create a new feature model by merging the others. We illustrate in Fig. 3 how the merge algorithm is used to build a simplified *source* FM. Each software artifact related to a source exists, and defines a feature (*e.g.*, *Picasa*, *FlickR*, *Twitter*). These features are modeled as independent FMs, and the merge algorithm is used to build the final FM that captures the associated variability.

Automated refactoring. Finally we now are able to manage the evolution of SPL FMs, but we still have to deal with the evolution of vocabularies (R4). FMs are not created and used by the same people, so we need to be sure that they all share the same vocabulary and if not, we have to give tools to modify this vocabulary: this need is directly linked to the agile principle which stands that every project needs to support *"face-to-face conversation"* with customers [12, principle 6]. Furthermore, in feature modeling the notion of vocabulary is essential, all their usage usually standing on feature names.

As SPL FMs are obtained by merging smaller ones, a change in a vocabulary may imply modifications in lots of FMs. The process of renaming each FM is still tedious and error prone. Moreover the usage of text editor operations, like regular expression replacement, is not always sufficient: a change in a domain vocabulary may imply to add a finer granularity or on the contrary to abstract things. Then we use a feature modeling language (Familiar [1]) offering scripting capabilities to automate these refactoring tasks. We are then able to rename a feature in a set, to add or remove features and so on, in a simple and explicit way.

5. MANAGING DEPENDENCIES

According to Bosch [5], the product line's success causes a significant increase in the number and complexity of dependencies among components in the SPL. Dependency management is thus essential to the consistency of a SPL and should not be an obstacle to its evolution. Our first solution to manage complexity of SPLs was to split its representation into several feature models corresponding to domain concepts. In a FM, constraints are then automatically deduced from the merging of domain objects FMs, making their maintenance transparent to the BS-Dev. However the domain concepts are themselves interrelated. It is therefore necessary to support the expression and management of these dependencies so that the agility gained is preserved.

5.1 Requirements

R5: Implicit modeling of dependencies and inferences. The domain modeling induces dependencies among domain concepts. For example, a renderer processing a particular type of information can only be linked to a source providing the same type of information. The SPL should take care of these dependencies to facilitate and ensure a rapid and consistent evolution of the SPL. Thus, BS-Dev in charge of renderers implementation focus on their artifacts, and the SPL tooling ensures that only adequate sources are connected with the renderers.

R6: Helping the BS-Dev to express dependencies. Some evolutions related to the integration of new domain objects require the management of specific dependencies (*e.g.*, adding a new source corresponding to a legacy information system may require the use of a dedicated renderer). Similarly, changing optional features of a concept may involve the establishment of new dependencies to guide the selections by BS-User (*e.g.*,, adding a title as a feature of some zone may imply to only accept renderers supporting such a feature). As it is necessary to handle dependencies between domain concepts, it is also necessary to allow the BS-Dev to express dependencies between specific domain objects.

R7: Enforcing SPL consistency. The number of composite configurations in which a domain object can be used is substantial. If it cannot be used in any composite configuration, it is useless and should not be part of the SPL. Verifying compatibility of a new domain object with all contexts in which it can be used, and more generally how changes impact the valid composite configurations is a challenge that we face.

5.2 Solution: Model Driven Evolution Process

It is known that FMs are not as expressive as metamodels: "*Feature models are well adapted to capture simple selections from predefined (mostly) boolean choices within a fixed choices within a fixed (tree) structure; and metamodels support making new structures by creating multiple instances of classes and connecting them via object references.*" [4]. In accordance with the identified needs and the previous observations, we choose to make explicit the relationships at model level. Their role is then to lead the configuration process. The proposed solution relies on *(i)* a metamodeling approach, *(ii)* the definition or restriction functions on models and finally *(iii)* the definition and verification of a realizability property to ensure the validity of the configurable products.

Metamodeling. To manage the relationship between FMs we have defined a metamodel that supports the definition of domain models as a composition of concepts corresponding to FMs and reifies the dependencies between them. These different elements support expression of cardinalities described in our case study. The metamodel is accessible from the project website[3]. Figure 1 is a simplification of a domain model conforms to our metamodel. This approach preserves a simple view of the product line from the BS-Dev viewpoint.

Restriction functions. Relations between the FMs lead the process of construction of composite configurations. Selecting a configuration in a FM can reduce the set of available selections in other linked FMs. To support these operations, *restriction functions* are associated to relations between FMs. A function is defined by a Familiar script [1] that makes explicit how FMs are related. These functions are defined by BS-Dev using composition operators between FMs. If the implementation of certain operators can be complex, it is hidden. Implementation and understanding of dependencies is then facilitated promoting the consideration of change. These functions meet the requirements R5 and R6.

A propagation algorithm based on these functions has been defined. It supports automatic propagation of user choices in building different contexts according to the cardinality defined at the level of models. For example, the choice of a source reduced locally all renderers that may be associated. This selection does not change the set of possible renderers for other sources. Conversely, the selection of some layouts may globally prohibit selecting some sources. When a source and a renderer are associated it is no longer possible to associate them with a different source or renderer.

Realizability checking. If the individual understanding of dependencies is simple, their composition can be very complex, making the strength of the approach. To ensure that the selection of a configuration does not prohibit the construction of a valid composite configuration (see R7), we check this *realizability* property [17] after each addition or modification.

A minimal kernel is initially created in which a composite configuration is always possible for all selected configurations. The FMs are then considered as sets of configurations. Adding or editing a set of domain objects thus implies to check the existence of valid configurations containing these domain objects. We proceed by a traversal of the graph corresponding to the domain model from those domain objects. The non-existence of a valid composite configuration leads to the refusal of the element.

6. CONCLUSION AND PERSPECTIVES

Feature-oriented modeling advocates the definition of feature models to support the variability of large software and SPL. However, the need for evolution in these feature models is not widely studied for now. In this paper, we used a large piece of software, started eight years ago and currently reengineered as a SPL, to illustrate such a need. Based on this example and the encountered evolutions, we focused on two points: *(i)* the evolution of domain objects captured in the feature models and *(ii)* the evolution of dependencies between the domain objects in the models. For each point, we expressed its requirements based on our *field experience* and proposed a solution to address these requirements. Our solutions rely on the definition of *composite feature models*, and the use of a *model-driven evolution process* to support it on large real systems.

One of the immediate perspectives of this work is to apply it to other software in order to strengthen its empirical validation. We plan to use SensApp[4], a scalable application used to collect data provided by sensors, supporting the definition of "Internet of Things" pieces of software. We

[3] `http://www.yourcast.fr/me12/`

[4] `http://sensapp.modelbased.net`

also plan to address several difficulties encountered while implementing composite feature models. First of all, keeping aligned the vocabulary used in the different models is challenging without support from the semantics domain. We plan to use ontologies and associated alignment mechanisms to support such a task [9, 11]. Another challenge we plan to address is how to maintain consistency between the different elements of a composite feature model and the associated configuration set. For example, adding a new constraint in the feature models may restrict the set of available configurations. How one can detect such a situation, and take accurate decisions with respect to the existing set of products already derived from the modeled product line?

Acknowledgments

The work reported in this paper is partly funded by the ANR YourCast project under contract ANR-2011-EMMA-013-01.

7. REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. A domain-specific language for managing feature models. In *SAC'11*, pages 1333–1340. PL Track, ACM, 2011.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Separation of Concerns in Feature Modeling: Support and Applications. In *Aspect-Oriented Software Development (AOSD'12)*. ACM, March 2012.

[3] Mathieu Acher, Patrick Heymans, Philippe Lahire, Clément Quinton, Philippe Collet, and Philippe Merle. Feature Model Differences. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE'12)*, volume 2012, pages 1–16, 2012.

[4] K Bak, K Czarnecki, and A Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. *Proceedings of the Third international conference on Software language engineering*, pages 102–122, 2010.

[5] Jan Bosch. Toward Compositional Software Product Lines. *IEEE Software*, 27(3):29–34, 2010.

[6] Jan Bosch and Mattias Högström. Product Instantiation in Software Product Lines: A Case Study. *Lecture Notes in Computer Science*, 2177:147–162, 2001.

[7] Ralf Carbon, Mikael Lindvall, Dirk Muthig, and Patricia Costa. Integrating Product Line Engineering and Agile Methods : Flexible Design Up-Front vs. Incremental Design. In *1st International Workshop on Agile Product Line Engineering / SPLC 2006*, 2006.

[8] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[9] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature Models are Views on Ontologies. In *10th International Software Product Line Conference SPLC06*, volume 1, pages 41–51. IEEE Computer Society, 2006.

[10] Sascha El-Sharkawy, Christian Kröher, and Klaus Schmid. Supporting Heterogeneous Compositional Multi Software Product Lines. *Proceedings of the 15th International Software Product Line Conference on - SPLC '11*, page 1, 2011.

[11] Martin Fagereng Johansen, Franck Fleurey, Mathieu Acher, Philippe Collet, and Philippe Lahire. Exploring the Synergies Between Feature Models and Ontologies. In *International Workshop on Modeldriven Approaches in Software Product Line Engineering MAPLE 2010SPLC10 Volume 2*, volume 2 of *SPLC'10 (Volume 2)*, pages 163–171. Lancester University, 2010.

[12] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development -San Francisco-*, Vol 9:28–35, 2001.

[13] Yaser Ghanam and Frank Maurer. Extreme Product Line Engineering — Refactoring for Variability : A Test-Driven Approach. *Processes in Software Engineering and Extreme*, 2010.

[14] Jianmei Guo and Yinglin Wang. Towards Consistent Evolution of Feature Models. In *Proceedings of the 14th international conference on Software product lines going beyond*, SPLC'10, pages 451–455. Springer-Verlag, 2010.

[15] Geir K. Hanssen and Tor E. Fæ gri. Process Fusion: An Industrial Case study on Agile Software Product Line Engineering. *Journal of Systems and Software*, 81(6):843–854, June 2008.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA). Technical Report CMU/SEI-90-TR-21, SEI, November 1990.

[17] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *RE'07*, pages 243–253, 2007.

[18] Sébastien Mosser, Gunter Mussbacher, Mireille Blay-Fornarino, and Daniel Amyot. From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models. In *10th international conference on Aspect Oriented Software Development(AOSD'11)*, , Porto de Galinhas, March 2011. ACM.

[19] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. Investigating the Safe Evolution of Software Product Lines. In Ewen Denney and Ulrik Pagh Schultz, editors, *GPCE*, pages 33–42. ACM, 2011.

[20] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *ICSE'09*, pages 254–264. ACM, 2009.