A methodological framework to enable the generation of code from DSML in SPL

Belarbi, Maouaheb

Published in:
Proceedings of the 22nd International Systems and Software Product Line Conference

DOI:
10.1145/3236405.3236426

Publication date:
2018

Document Version
Peer reviewed version

A methodological framework to enable the generation of code from DSML in SPL

Maouaheb Belarbi
University of Namur
Namur Digital Institute
PReCISE Research Centre
Faculty of Computer Science
Namur, Belgium
maouaheb.belarbi@unamur.be

ABSTRACT

Software Product Line has acquired a significant momentum at the end of the 1990ies since it allows the production of variable software systems corresponding to the same domain portfolio. The effectiveness of the derivation process depends on how well variability is defined and implemented which is a crucial topic area that was addressed among two essential trends: On the one hand, starting from Domain Specific Modelling Language to express domain requirements and automate the code generation with Model-Driven Engineering techniques and on the second hand, exploiting the soar of variability mechanisms.

In this context, the current research presents a method that unifies the two aforementioned approaches to cover the overall strategies by defining a framework that allows a better code generation in terms of documentation, maintainability, rapidity,etc. The starting point is the usage of the Domain Specific Modelling Language to represent the stakeholders requirements. Then, the resulting meta-model will be converted into one our several Feature Diagrams on which variability mechanisms can be applied to generate all the family products.

A preliminary experiment has been undertaken to design the methodology of the proposed software factory in a meta-model. The validation task was evaluated with an academic use case called HandiWeb developed to facilitate handicap persons access to the internet. The first results allow us to put the hand on the key challenges that must be resolved by the proposed methodology.

CCS CONCEPTS

• **Software and its engineering → Software design engineering**;

KEYWORDS

DSML, SPL, methodology, software factory, variability

SPLC'18, 10–14 September, 2018, Gothenburg, Sweden

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123

ACM Reference Format:
Maouaheb Belarbi. 2018. A methodological framework to enable the generation of
code from DSML in SPL. In *Proceedings of 22nd International Conference on Software Product Line (SPLC'18).* ACM, New York, NY, USA, Article ?, 8 pages. https://doi.org/10.475/123

1 INTRODUCTION

Software Product Line (SPL) engineering has acquired a significant attention at the end of the 1990ies as an emerging paradigm for producing similar systems branching from a domain portfolio. In other words, SPL governs a family of software products that share common set of features, which represent a logical unit of behavior specified by a set of functional and quality requirements, and they differ each other according to variable aspects. This last, was defined as the ability of a software system or software artifacts to be extended, customized or configured to use in a specific context [23]. Thus, variability specifies parts of the system to be kept variable and not fully defined during design phase, which allows a fluent development of its different versions.

The effectiveness of a SPL approach is close fitting to how well variability is implemented and managed. However, as the complexity of such variability grows over time, variability design and implementation have become a challenging issue. One of the key steps is to select the variability implementation mechanism or a combination of mechanisms that is suitable for a given system. The notion of variability is a crucial topic area to tackle that was addressed in [14]. At this level, two trends are of interest: On the one hand, starting from a Domain Specific Modeling Language (DSML), how to automate the code generation with Model-Driven Engineering (MDE) techniques [20], and on the second hand exploiting the soar of variability mechanisms.

The discussed variability mechanisms describe how variable features are implemented on design and implementation phases using different programming techniques. In this paper, we consider the categorization mentioned in [8] to discuss the following mechanisms: Cloning, Conditional Compilation, Conditional Execution, Polymorphism, Module Replacement, Aspect Orientation and Frame Technology. Indeed, these latter variability mechanisms vary in different parameters such as techniques, binding times, granularity. Consequently, the aforementioned mechanisms are applied, in practice, in various development scenarios. However, the required flexibility

and adaptability of SPL necessitates a high customization of products family according to stakeholders' requirements. In industrial case studies, each mechanism showed advantage but brought challenges during each software development. Furthermore, these mechanisms do not present a methodology to manage their aggregation and enhance the self-adaptation of an SPL to variable contexts.

In another area, Model-Driven Engineering (MDE) was integrated in SPL to facilitate variability implementation and management [12]. MDE techniques rely on the DSML meta-model that describes the main concepts of the domain and their relations. MDE allows system designers (DSML end-users) working closer to the system domain as they will manipulate concepts from the real system [10]. Finally, generative methods can transform DSML into code artifacts. DSMLs offer greater expressiveness in expressing the stakeholders requirements within an applicative domain. Indeed, DSMLs can offer graphic and/or text notations specific to domains and can naturally encompass the expressivity of general purpose modeling language. However, the benefits of DSMLs do not come for free, as the language abstractions and tools to automate development need to be first developed and later maintained. Research claims that it is costly and hard to define modelling languages with tool support; that domain-specific languages can be created effectively only when the domain does not change; and that Model Driven Development (MDD) does not scale [15]. Here, the evolution of meta-models corresponding to the employed DSML implies the evolution of the developed generating scripts which is highly complicated.

In this light, our contribution unifies the two aforementioned trends by making the use of their gain and avoiding their boundaries. Hence, the starting point consists in transforming a DSML into a Feature Diagram (FD) (or possibly several FDs), which allows exploiting variability mechanisms further. Besides, it is crucial to dispose a methodology that covers all the elements the product family is built from as well as their corresponding composition rules. This methodology clarifies how the various parts of a selected configuration may be combined. In this context, the objective of the current research is presented in a method that aims to:

(1) Cover the overall strategies to generate the variants in the solution space that corresponds to a problem space depicted in a Feature Diagram;
(2) Present a methodology to guide the software engineers during the definition of a generative strategy;
(3) Generate the overall artefacts corresponding to the selected features and according to the selected strategy

The rest of the paper is organized as follows: Section 2 introduces some of the relevant related work. In section 3, we present the research questions mined throughout this thesis followed by a description of the proposed research methodology in Section 4. In section 5, we give a shade of light on our preliminary results. Finally, we present the work plan for the current year in Section 6 followed by a conclusion in Section 7.

2 RELATED WORK

In this section, we discuss the existing variability implementation techniques. To do so, we start with variability mechanisms in SPL and then we discuss integrating MDE techniques that employ DSML to model and derive product variants.

2.1 Variability mechanisms

Most of existing implementation mechanisms found in the literature were usually classified as compositional or annotative approaches [8]. Compositional approaches implement features as distinct code units. In order to obtain a product line part for a feature selection, the code units are determined and composed usually at compile-time or deploy-time. Whereas, annotative approaches implement features by planting implicit or explicit annotations directly in the source code. The prototypical example is the use of `#ifdef` and `#endif` statements of the C-preprocessor to surround the target code.

Clone-and-own: The easiest way of producing a variant of a certain software product consists in copying a code or non-code artefact and evolving it further without keeping connection with original version. Cloning is perceived to be a simple reuse mechanism that saves both time and resources. It allows software engineers to start development from implemented and verified artefacts. Moreover, the changed introduced to the cloned do not threat old variants. However, cloning is technically a new branch that can be used to develop a new product. This leads to different problems, especially for software maintenance. Performed modifications in the main product need to be merged into all other product variants and vice versa. This additional merging process is high cost and cannot scale for a large number of products.

Conditional Compilation: It is one of the most widely used techniques discussed in [18] and characterized by using preprocessors. The pillar principle is to conditionally include or exclude variability with `#ifdef` annotations. In fact, most software developers are familiar with the easy-to-use preprocessor directives. Moreover, this implementation mechanism allows for fine-grained variability without involving architectural overhead. The enclosed fragments of code surrounded with `#ifdef` blocks can be either classes, functions, type declarations, etc. Furthermore, variability is not separated from the whole code and is explicit to identify. Although, conditional compilation mechanism show some clear benefit, they received severe criticism because of some boundaries. Indeed, the code is usually complex and hard to understand since `#ifdef` statements are nested with a complex structure.

Conditional Execution: In [11], it is considered as an another annotative approach that aims to realize variability by coding it explicitly using conditional if-else code parts. Conditional code is enabled or disabled depending on the given parameters and the program behaves differently at runtime. In case of multiple variation points, if-else are replaced with switch blocks. An obvious benefit of this mechanism is that it is easy to use with no learning effort. Moreover, variabilities are instantiated at runtime. Therefore, it provides

high flexibility to adapt the system to unforeseen requirements. For this reason it is considered by Bosch et al. [13] as a promising trend in software variability, although this introduces complexity and other side effects. Despite the benefit mentioned above, developers have encountered several challenges: Variants must be fine-grained to be implemented as conditional code blocks. In addition, it is hard to distinguish between variation logic and code functionality because they are nested together. Finally, the compilation speed and variation at run-time are degraded because of the inclusion of all variant elements from code compilation until running.

Polymorphism: In [1], three types of polymorphism are in wide use: Subtype Polymorphism, Parametric Polymorphism, and Overloading. While Subtype Polymorphism and Parametric Polymorphism occur at either compile-time or run-time, Overloading is often used at compile-time. For example, when an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time. As a compositional approach, variability is separated in different files from common features. In addition, this mechanism allows a high flexibility for run-time variability. Finally, the open variation is frequently ensured by Polymorphism to enable developers to extend framework capabilities without modifying already compiled framework code. However, the adoption of Polymorphism in practice presents several challenges since it increases the risk of software defects at run-time errors such as illegal pointers.

Aspect-Oriented-Programming (AOP): This mechanism relies on code weaving techniques that require external tool support such as AspectJ [6]. As a compositional mechanism, it has the benefits of the separation of common and variable features into separate files. Depending on the aspect weaver, variability can be resolved at both compile-time and run-time. Despite the benefits above, AOP is not supported by a programming language and therefore it is difficult to apply it rapidly in development process without learning efforts. Moreover, Kastner et al.[9] have conducted a case study to refactor variability realizations in Berkely database system using AspectJ, which turned out to be unsuitable for implementing variable features. Similarly, a literature review established by Amine et al. [2] shows that the major challenges of AOP are the increase in code size and low code comprehensibility.

Frames technology: Basset[5] introduced the frames technology that uses adaptable code frames and assemble them by a frame processor. It showed several benefits like other compositional approaches. However, Variation points are explicitly identified in code and variation elements are only handled as textual. Although, this mechanism is not widely used because it requires special tool support and the standard syntax is still missing.

Recent research regarding implementation mechanisms has led to comparison summarized in [7]. This last, presents a characterization of the variability mechanisms derived from the aforementioned categorization. The discussed mechanisms differs according various aspects. Initially, variabilities are instantiated and bound to a variant at a specific point of times in example at construction time or running time. Here, we confess that mechanisms with early binding time resolve the variability configuration space early and optimize running efficiency. However, approaches with late binding time ensure dynamic system adaptions. In addition, traceability describes the ability to track a feature among functional code. Compositional approaches supports better traceability since code is fragmented into separated units. In contrast, annotative approaches handles poorly this characteristic seen that variability is scattered over source code. Moreover, modularity is a needed characteristic for modular reasoning or even separate compilation. It is possibly supported in some compositional approaches. For example, when using components, plug-ins, sub-jects or hyper-modules this is well handled. All the variants of the SPL should be syntactically correct and well-typed. Finally, compositional approaches ensured variants safety thanks to their composition mechanism. However, annotative mechanisms can easily generate incorrect variant elements.

2.2 MDE techniques

In traditional SPL engineering approaches, variability is mainly handled using the aforementioned mechanisms and performed on a FD. Despite their wide acceptation, there is no de-facto standard. Hence, recent efforts have proposed to facilitate variability implementation, management and tracing by integrating Model Driven Software Development MDSD techniques [16]. It is a natural candidate to fit in the general framework of SPL. In fact, MDSD improves the way the software is developed by capturing key features in system models. In contrast to traditional modeling, MDSD models may comprehend arbitrary complex systems, their rationales and requirements, as well as the domain and technical constraints while being processed by tools [21]. In this context, DSML is developed by specifying the domain meta-model in order to identify all the concepts pertinent for the creation/ generation of an application. Transformation rules are defined to generate code from the application model (when it will be available). Obviously, the future application model must be conformed to the meta-model depicted via DSM. In addition, manually written source code can also be created that are intended to be combined with the automatically generated code later. The goal is here to maximize the reuse of all the asserts developed during the liftetime of the software factory. DSML is characterized by its expressiveness potential. Indeed, variability is well defined in both problem space and solution space as well as the mapping between them is standardized and automated via transformation rules. However, FD is commonly more used to model SPL variability and define compactly all features. Besides, the desired configuration of variant products is performed on a FD and consequently it is of our interest to profit not only from DSML expressiveness but also from the feasibility of FD.

2.3 Discussion

To recapitulate, the first part of this section discussed variability mechanisms. We found that existing tactics aim to handle this variability according to stakeholder requirements. In fact, a resulting software product is generated according to a selected configuration. That means the selected features are implemented corresponding to a binding time, a granularity, an explicit or implicit variation point and open/closed variations. In SPL, the dynamic assembly of products increases the flexibility of product configuration options and is often preferred for that reason. In critical systems, new tendencies were born to permit the possibility to switch between various systems modes automatically. So, to allow a smooth transition among different binding times. For example, runtime concerns and post-deployment configurable options grab the attention of an important research work quota. Hence, to overcome this trouble, software engineers have been trending toward relying on new methods and representation techniques able to support variability at any time, any granularity varying context conditions. To the best of our knowledge, limited research have proposed a methodology that allows the combination of variability mechanisms to generate a selected configuration. Upon this, the objective of the current research is to define a methodology to design a Software Factories that aim to cover all the generation code tactics to realize all software variants. To do so, our starting point is DSML to well-define the variability and optimize the expressivity of the requirements concerning the domain. This model would then be annotated with meta-information to express the requirements about the binding times, the open/close status of the variants and the rationales of these decisions and further the resulting models will be converted into FDs to make use of the SPL variability techniques.

3 RESEARCH QUESTIONS

Taking into account the main research focus of this Doctoral thesis project, we present in this section the planned research questions. The realization of the proposed methodology goes through the following steps:

(1) How to transform a DSML specification into FDs?
   - DSML raises the level of abstraction by specifying software systems directly by domain concepts. Thus, DSMLs can offer greater expressiveness than FDs in expressing the requirements of stakeholders within an applicative domain. Indeed, DSMLs can offer graphic and/or text notations that are specific to domains and defined with more expressive meta-languages (i.e. meta-modelling or ontological languages) and do not suffer from constraints related to FD (i.e. hierarchical organization of features). However, since simplifying assumptions in FDs can significantly reduce the complexity of the transformational approach to code, this thesis will use FD as an intermediate step in the path to code generation from a DSML.

(2) How to transform a configuration expressed in an FD into code source that respects the chosen tactics?
   - A feature is a characteristic or end-user-visible behaviour of a software system [17]. Many tactics exist to transform a feature into code (AOP, metaprogramming, IoC, patterns,...). But the discipline lacks methodologies in this regard to usefully guide the method engineer in designing a strategy to assemble these tactics in a coherent and useful way. Different points of view must indeed be taken into account when designing a "software factory" guided by configurations: correction, feasibility, system maintenance,return on invest, ease of use, skill level, complexity, maintenance and scalability of the SF itself...
   - How to interpret features usefully to make the generative approach possible.
   - How to document tactics so that they can be assembled into a strategy
   - How to make a strategy operational?

(3) How to evaluate the relevance of a generative strategy? The relevance describes how well the generative strategy satisfies the expectations of the developers in terms of cost, maintainability, rapidity, documentation, etc.
   - what is the cost of a strategy?
   - how easy is it to maintain a strategy?

4 RESEARCH METHODOLOGY

The objective of this thesis is to propose a methodology to design Software Factories that cover the possible infinity of generation code strategies to derive product variants belonging to a FD. Note that our starting point is to well define variability through DSML that will be converted later into a FD. Hence, our thesis is performed through the following stages:

The first step consists in going over MDE techniques and how they were integrated in SPL to profit from DSML expressiveness. However, FD is perhaps the most common formalism used to model SPL commonalities and variabilities [19]. Therefore, we need to convert the DSML to FD to take advantage of the variability implementation techniques specific to this family of languages. MDE may provide interesting tools to ensure transitions from models to models or even code. We also feel the need to extend the meta-model of the domain with extra meta-information specific to our methodology in order to usefully guide the transformation process. Hence, the transformation task requires a clear understanding of the abstract syntax and the semantics of both the extended DSML as the source model and FD as the target. As a result, we must on the first hand deepen our knowledge about the different FD meta-models created throughout other research works, besides the various model transformation tools and, on the second hand, see how the meta-modelling language used to design the DSML can be extended in order to provide the extra-information required to guide the transformation process (i.e. which concepts are denoting features, which preferred path among the concepts a transformation must follow to generate the hierarchical representation of the features...).

The second step consists in reviewing variability implementation mechanisms to have solid pillars on the existing approaches as well as the existing frameworks and tools to aggregate a combination of mechanisms. Hence, we can identify the parameters and methods of aggregation, their benefits as well as their challenges. Besides, we identify variability types that can influence later on the generation code process. For example building time and granularity are considered as crucial parameters. This leads us to discuss the strengths and weaknesses of the implementation mechanisms according to the variability characteristics in order to guide the enhance generation code process.

After that, we can design the overall architecture that includes the conversion task of the DSML into FD by model transformation and the derivation of the product variants according to the strategy that has been selected among the different possibilities.

The validation process can be realized in an applicative domain to prove the validity and performance of the proposed contribution. To do so, we will rely first on an academic use case called HandiWeb developed to assist disabled persons while using the internet services. This proof-of-concept should help us convince then companies of the validity of our approach in order to deploy it on a larger scale industrial case study.

5 PRELIMINARY RESULTS

The preliminary results were conducted to define the general methodology of the proposed Software Factory. Our early research have led to the meta-model depicted in Fig. 1.

The principal class of the proposed metamodel are *Feature Diagram* (FD) that denotes main entry of the FD metamodel, *Strategy* which represents the *Selected Tactics* that can be used to implement *Configurations*, and finally *Configuration* that collects a set of *Selected Features* within a *feature model*. A *strategy* denotes for a subset of *features* the *tactics* "on the shelf" that are eligible for its implementation, with attributes that provide the rationales. In a FD, some *features* may have a *variant time* that is defined once and for all while others (in *Selected Feature*) are defined in the context of a peculiar *configuration*. *Tactics* denote general mechanisms that can be used with *asset types* to produce customized core components of the target system from *core assets* identified by an *URI*. *Scenarios* [3, 4] document the rationale of the *variant time* of each feature in a configuration in order to guide the temporal assembly of tactics and the methodology.
At this level we can define the first step of the proposed methodology as the identification of the desired configuration through the selected set of features.

Besides, to illustrate the importance of defining feature binding times, we consider the example of Handiweb application depicted in Fig. 2 where the visual handicap can be handled at compile-time or run-time. Thereby, this leads to two different applications. The first one is especially developed for disabled people and thus offers an evolved behaviour with advanced techniques. For example, haptic touch-screen, matrix pins and more advanced technologies are available today. The second type of application, as shown in this paper, tries to adapt its exploitation for disabled persons at running time. Thus, it provides the basic behaviour to make the application adapted for visual impairments.

The second step of the proposed methodology consists in selecting tactics to implement the desired configuration according to the feature variation times and the already defined non-functional properties.

Initially, the tactics are characterized by their binding time, cost and required engineering skills to make them useful. Some variability mechanisms allow resolving variability at run time, i.e. Conditional Execution which is easy to use with no learning effort whereas AOP requires engineering skills.

Finally, the proposed methodology sorts available implementation tactics that are suitable for the generation of a selected configuration according to their cost and required skills. For instance, it does not allow developers to select Conditional Execution tactics to implement a feature specified at compile-time. In other words, the selected tactics must be coherent to assemble.

The preliminary experiments were conducted to implement HandiWeb according to the proposed methodology. The usage scenario of the application is as follows: A user must select the service of search or shop as a mandatory feature. The search operation is ensured by Google engine or Yahoo engine while the purchased goods are chosen via Amazon or EBay. Furthermore, Handiweb opens its door to handicapped persons by configuring mandatory services according to the disabled type of the current user. According to this configuration, the chosen generative strategy by the developer is as follow: The searching operation is implemented with AOP, while the shop is realized with the design pattern Factory. Both of the aforementioned features are resolved at compile time. Finally, the optional handicap functionalities are developed at run time via the conditional execution mechanism.

To reveal the complexity hidden behind the simple handiWeb FD, the number of generated variants is over 90. In practice, if we consider a bigger FD the resulting amount of variability, if not well treated, will be blocking. Hence, this problematic entails the need to possess a methodology that allows variants generation and their adjustment for the plethora of existing contexts.

- As shown in Fig. 2, the selected configuration includes visual handicap features at run-time and search service resolved at compile-time.
- At this level, according to the aforementioned methodology the second step consists in selecting the applied variability mechanisms to implement chosen features. The search operation was implemented with design pattern Decorator by including Jsoup library. While the blind feature was set up with Conditional Execution to replace the button and text by mouse events and aural messages. The same feature can be implemented with other advanced techniques using Aural User Interfaces or haptic aural touch-screen...

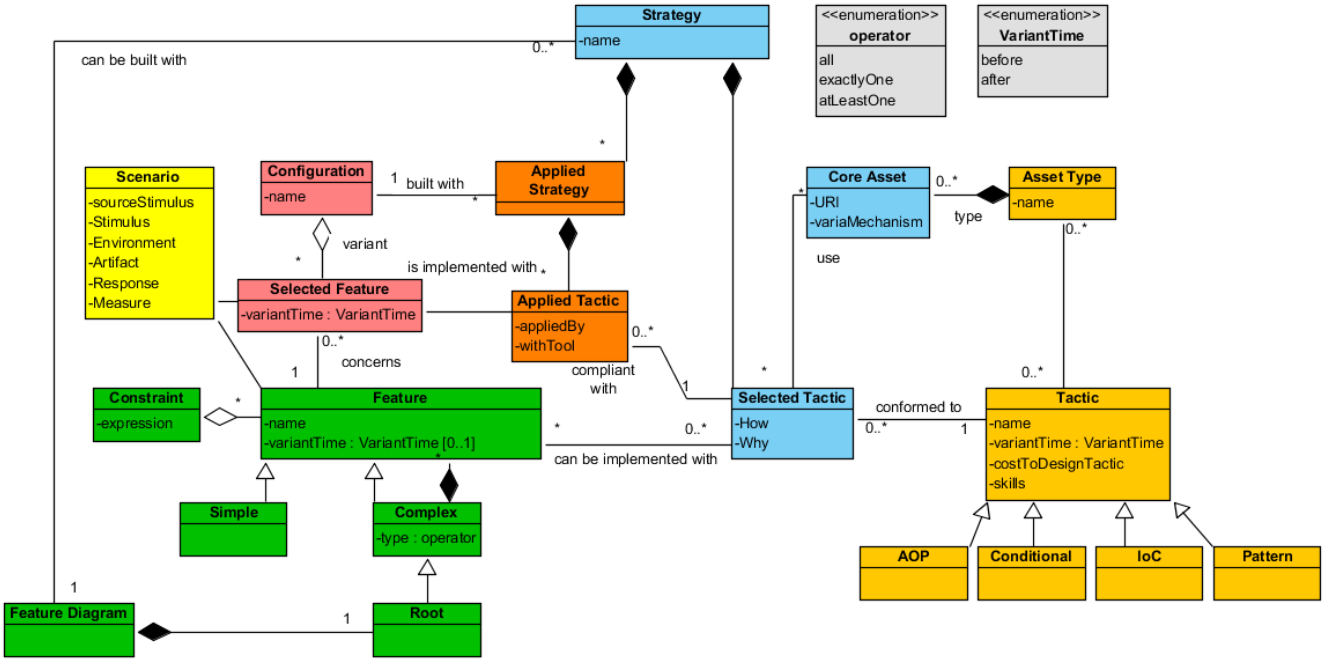

Figure 1: The metamodel corresponding to the proposed methodology.

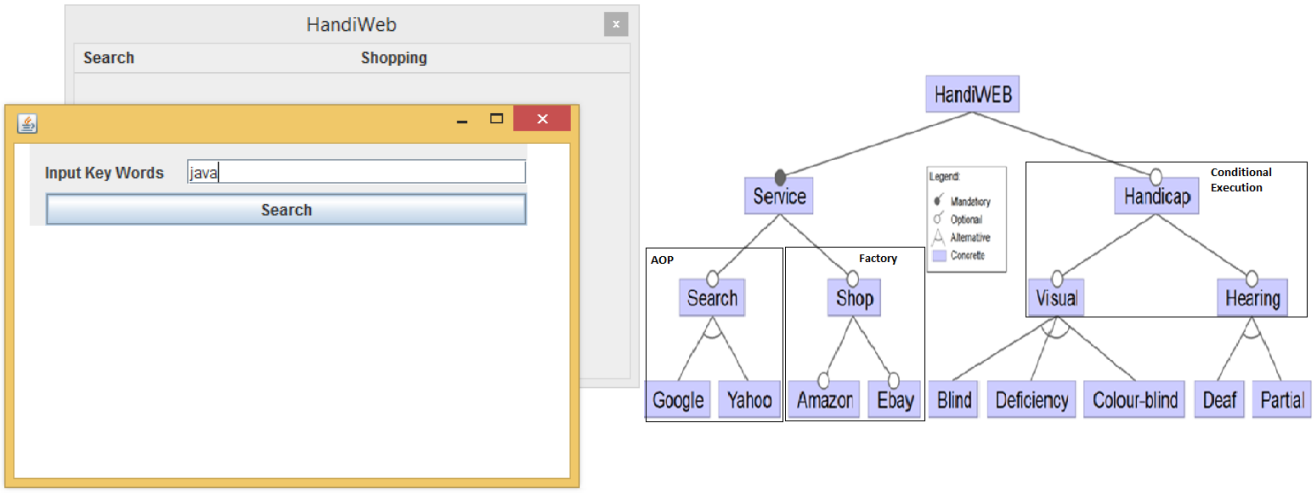

Figure 2: Left figure: HandiWeb application. Right figure : Feature Diagram corresponding to HandiWeb.

- Finally, The generation of the desired configuration code source is performed with featureIDE plugin. This last, is an open-source framework for feature-oriented software development (FOSD) based on Eclipse[22]. FOSD is a paradigm for construction, customization, and synthesis of software systems. Code artefacts are mapped to features and a customized software system can be generated given a selection of features. As we can see, there is a distinction between the selected tactics that implement the selected features independently, on the one hand and the applied tactic to generate the whole of selected configuration.

The preliminary experiments allow us to put the hand on the key challenges that must be resolved by the proposed methodology. First, selecting the right set of features from the overall available feature in FD is delicate because:

- The multiplicity of stakeholders' functional requirements;

- The positive or negative impact on the non-functional properties;
- The desirable non-functional properties of the final software product.

The existing variability mechanisms do not consider stakeholders requirements and constraints especially regarding non-functional preferences. Consequently, the designed Software Factory must employs an artificial intelligence planning techniques to automatically select suitable features that satisfy both stakeholders' functional and nonfunctional preferences and constraints [3].

Second, the input FD notation must be completed with possible binding times corresponding to each feature as well as cardinality.

Finally, the selection of applied tactics must be documented with additional information. For example, the selection of Conditional Compilation mechanism to adapt HandiWeb for blinds should be documented by its position in the source code that is already generated with a different tactic.

6 WORK PLAN

Figure 3 shows a twelve-month work plan to visualize advances and activities to proceed as a half-time researcher. Concretely, the tasks are the following:

Task 1. DSML conversion into FD. Until August, we plan to contribute the conversion of DSMLs belonging to a domain portfolio into a FD by the mean of MDE techniques. Hence, it is necessary to sweep MDE methods that allow the transition from a model to another and customize them according to the specificities of FD.

Task 2. Variability Systematic Literature Review. Until November, we need to have solid pillars on SPL variability to be able to identify key factors that influence on product variants derivations. Hence, we need to perform a systematic literature review for variability and implementation mechanisms.

Task 3. Operational combination variability mechanisms. Until February, we intend to investigate existing approaches that allow aggregating variability mechanisms, their advantages as well as their deficiencies. From that point, we identify an operational strategy aware of the aforementioned variability key factor.

Task 4. Transforming a selected configuration into a FD to computer assets. Until May, we need to look for how to transform FD to computer assets by exploiting MDE techniques. Functional and non-functional features should be recognized during the transformation process as well as binding times corresponding to each variation point.

Research work must be submitted to the scientific community in order to get a valid feedback and provide visibility. Thus, a publication plan was defined with two types of expected publications: (i) conference proceeding papers and (ii) workshop papers. Indeed, conferences and workshops provide an excellent environment to get feedback from experts in the domain and improve the thesis contributions by reviewers comments. A detailed list of the expected paper submissions is presented via Table1.

7 CONCLUSIONS

The derivation of SPL variants is a practical challenge that entails extra efforts to reduce development cost. Indeed, two trends are of interest: On the one hand, starting from DSML how to automate the code generation with MDE techniques and on the second hand exploiting the soar of variability mechanisms.

This paper presents the ongoing Ph.D. thesis research which tend to propose a methodology for designing a software factory able to cover the all over strategies to derive product variants. The proposed methodology starts by defining variability with DSML and then convert it into FDs on which we can exploit variability implementation mechanisms.

In future work, we plan to perform the aforementioned activities in the work plan starting with the conversion of DSML into FDs.

REFERENCES

[1] A. Demaille A. Duret-Lutz, T. Geraud. 2001. Design Patterns for Generic Programming in C++. In *6th Conference on Object-Oriented Technologies and Systems, 2001.*

[2] A.Kamil A.Fazal and A.Oxley. 2010. A review on aspect oriented implementation of software product lines components. *Information Technology Journal* (2010).

[3] Felix Bachmann and Paul Clements. 2005. *Variability in Software Product Lines.* Technical Report CMU/SEI-2005-TR-012. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675

[4] Len Bass. 2013. *Software architecture in practice.* Addison-Wesley, Upper Saddle River, NJ.

[5] Bassett and G.Paul. 1996. *Framing software reuse: lessons from the real world.* Prentice-Hall, Inc.

[6] S.Duszynski B.Zhang and M.Becker. 2016. Variability mechanisms and lessons learned in practice. In *Variability and Complexity in Software Design (VACE), IEEE/ACM International Workshop on, 2016.*

[7] C.Gacek and M.Anastasopoules. 2001. Implementing product line variabilities. In *ACM SIGSOFT Software Engineering Notes, 2001.*

[8] CKastner and S.Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE), October 19-23, 2008.*

[9] S.Apel C.Kastner and D.Batory. 2007. A case study implementing features using AspectJ. In *Software Product Line Conference SPLC, 2007.*

[10] X.Cregut F.Zalila and M.Pantel. 2013. A transformation-driven approach to automate feedback verification results. In *Proceedings of the Third International Conference on Model and Data Engineering, 2013.*

[11] Thomas Thum Thomas Leich Fabian Benduhn Gunter Saake, Jens Meinicke and Reimar Schroter. 2017. *Mastering Software Variability with FeatureIDE.* Springer.

[12] H.Papajewski I.Groher and M.Voelter. 2007. Integrating Model-Driven Development and Software Product Line Engineering. In *Eclipse Summit 07: Proceedings of the Eclipse Modeling Symposium, 2007.*

[13] R. Capilla J. Bosch and R. Hilliard. 2015. Trends in systems and software variability. In *IEEE Software, 2015.*

[14] D. Hoffman J. Coplien and D. Weiss. 1998. Commonality and variability in software engineering. In *IEEE Software, Nov/Dec 1998.*

[15] J.Tolvanen and S.Kell. 2016. Model-driven development challenges and solutions - experiences with domain-specific modelling in industry. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, 2016.*

[16] C.Kim P.Hwan S.Lau K.Czarnecki, M.Antkiewicz and K.Pietroszek. [n. d.]. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference*

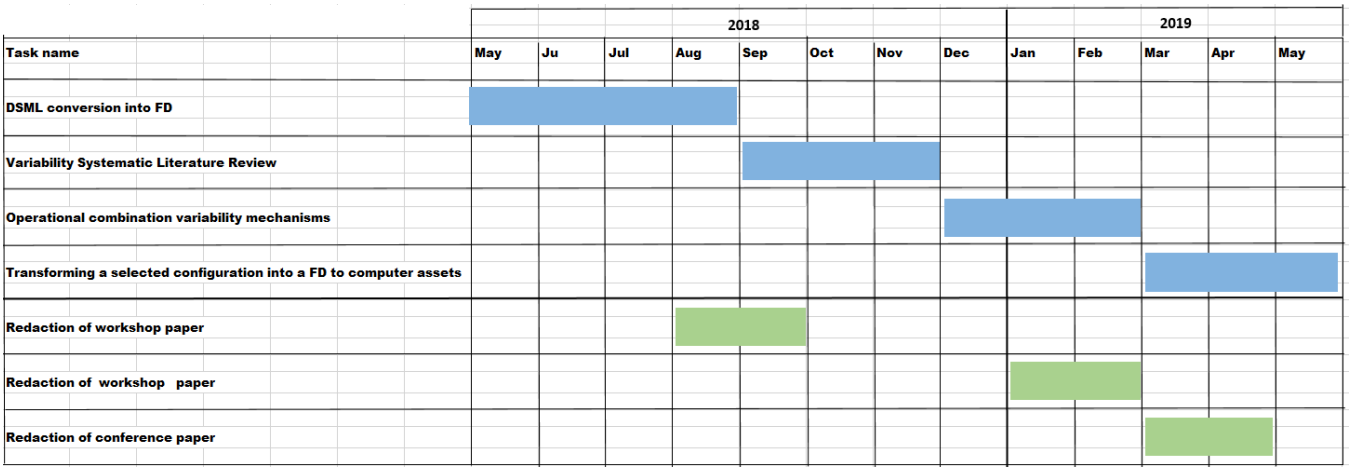

Figure 3: The detailed work plan for the 12 months Ph.D. studies.

Table 1: Publication plan

Publication scope	Conference/Journal	Deadline
`Objective 1 (workshop paper)`	VAMOS 2019	October 2018
`Objective 2 (workshop paper)`	MODELS 2019	March 2019
`Objective 3 (conference paper)`	SPLC 2019	May 2019

on Object-oriented programming, systems, languages, and applications, 2005.

[17] N.Cardozo K.Mens, R.Capilla and B.Dumas. [n. d.]. A taxonomy of context-aware software variability approaches. In *Companion Proceedings of the 15th International Conference on Modularity, 2016.*

[18] M. T. Valente M. V. Couto and E. Figueired. 2011. Extracting software product lines: A case study using conditional compilation. In *IEEE Computer Society, 2011.*

[19] F.Fleurey P.Lahire S.Moisan M.Acher, P.Collet and J.Rigault. [n. d.]. Modeling context and dynamic adaptations with feature models. In *4th International Workshop Models@ run. time at Models, 2009.*

[20] M.Voelter and I.Groher. 2007. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference, September 10-14, 2007.*

[21] I.Galvão J.Noppen S.Khanand H.Arboleda A.Rashid N.Anquetil, B.Grammel and A.Garcia. [n. d.]. Traceability for model driven, software product line engineering. In *ECMDA Traceability Workshop Proceedings,2008.*

[22] F.Benduhn J.Meinicke G.Saake T.Thüm, C.Kästner and T.Leich. [n. d.]. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming, Elsevier, 2014* ([n. d.]).

[23] J.Bosch Van Gurp and M.Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Working IEEE/IFIP Conference on Software Architecture, August 28 - 31, 2001.* pages 45–54.

[24] T. Berger S. Duszynski M. Becker and K. Czarnecki Y. Dubinsky, J. Rubin. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings of the Conference on Software Maintenance and Reengineering, 2013.*