Feature Model to Product Architectures:
Applying MDE to Software Product Lines

Abstract

A Software Product Line (SPL) is a portfolio of products that targets a particular domain. Feature Models are generally used for modeling domain knowledge including variability within SPLs. The Product Line Architecture (PLA) defines the structure that all potential products in the SPL share. Designing a good PLA is challenging since different products may require different characteristics, and it is difficult to achieve an acceptable trade-off. In this paper we apply Model-Driven Engineering techniques for systematizing the Domain Engineering stage to enable the automation of the Application Engineering stage. We use features to modularize architectural decisions and we encode them as model transformations that render the fragment of the product architecture that addresses the features. Then, we make the rationale explicit, and we enhance evolvability and incrementality diminishing design complexity. Product implementation is derived by means of generators analogously. We show our approach by developing a Meshing Tool SPL.

1. Introduction

A software product line (SPL) is a set of software systems that share a common and managed set of characteristics that satisfy the needs of a particular market segment or mission, and that are developed from a common set of core assets following a planned process [6]. Traditional SPL development processes identify two main technical stages: Domain Engineering, where reusable assets are developed and maintained, and the scope and production plan are defined, and Application Engineering, where particular product requirements are gathered, and the product is built by arranging the reusable assets according to the production plan. There are several SPL development processes that follow this structure [6][8][29]. The Domain Engineering stage consists of three activities: Domain Analysis, Domain Design, and Domain Implementation. The Domain Analysis produces the Domain Model that gathers all potential features. This model is used during Domain Design as a basis for designing the Product Line Architecture (PLA) and the catalogue of software components that will populate it. In the Domain Implementation, the designed components are built. Both, the Domain Model and the PLA capture commonalities and variabilities of the SPL. The PLA is built in such a way that it articulates all the identified features according to the quality requirements for the whole SPL.

Counting on a Domain Model, a PLA and a component catalogue, the Application Engineering stage consists of choosing the desired features, eliminating variabilities from the PLA yielding a Product Architecture (PA), and selecting the appropriate component implementations. These components are put together according to the PA obtaining the desired product.

Feature models are a widely used approach to domain analysis capturing commonalities and variabilities [10], and there are several methods and notations for generating them [1][18]. But still having a well documented domain analysis, the architect has a huge responsibility on the success of the SPL: if the PLA he/she designs is not appropriate, all products will be flawed. This makes the PLA design a hard task. What is even worse, the PLA may become inadequate if the SPL scope evolves since the rationale of the PLA design is usually lost in the design process, and the PLA should be redesigned from scratch [11].

Several authors have proposed approaches to relate feature models and product architectures. A good review can be found in [16]. For Berg et al. [3], approaches for variability management relating requirements and solutions should be consistent, scalable, traceable and should provide visualization media. They state that these goals are difficult to achieve because, as there is no standard for feature modeling, their consistency could not be enforced, and also that as models grow they become complex and thus they become non-scalable and non-traceable either. Visualization is the
only aspect fulfilled by feature models. In our approach we achieve evolvability by incrementally building the product architecture, and we make the design process consistent and traceable by using feature-to-architecture transformation rules.

There have been some successful experiences in automating product generation in SPLs [15][21], most of them targeting specific domains. Automation is thus desirable and also feasible.

In this work we apply Model-Driven Engineering [27] techniques to provide a systematization of the Domain Engineering stage so as to enable the automation of the Application Engineering stage. To this end, we consider features in the Feature Model to represent functional areas, and architectural decisions are explicitly recorded as a set of model transformation rules attached to each feature or cohesive set of features. These rules express how to build the fragment of the PLA that provides the functional areas that take care of the given features, and that organizes the identified components in a way that they address the quality attributes impacting the features, provided that certain quality attributes. As remarked in [6], some of these quality attributes such as security, performance, or maintainability have a direct impact on the architecture that addresses them. Also, for each component populating the PLA either an implementation or a generator is implemented during Domain Implementation. Therefore, for generating a particular product it is enough just to define a particular Feature Configuration Model. The actual Product Architecture is automatically generated by applying the set of rules of the selected features. The actual product is built by using the implementations and by applying the generators of those components participating in the Product Architecture. If the Domain Model happens to evolve, the model transformation rules and components and/or generators of the impacted features need to be created or updated, and they become available for future product instantiations without having to redesign the whole PLA.

The rest of the paper is structured as follows. Section 2 presents the defined development process, explaining its rationale. Section 3 details the activities and artifacts involved in the Domain Engineering stage, and Section 4 covers the Application Engineering stage. We successfully applied this process to develop a Meshing Tool software product line$^1$, and report a key part of this experience in Section 5. Related work is reviewed in Section 6. Finally, Section 7 concludes with a discussion and suggests further work directions.


Figure 1. Model-Driven Development Process of Software Product Lines.

2. Model-Driven Development of Software Product Lines

In this work, we apply MDE to refine the traditional approaches to SPL. We deal with the Domain Engineering and Application Engineering stages, both organized in terms of three major activities: Analysis, Design and Implementation. Analysis uses features to explicitly define functionality, and also variability in the case of Domain Analysis. Design is architecture-centric, and tackles critical structural and quality-addressing decisions. Implementation deals with actual component and product development. Figure 1 illustrates the organization of the proposed process and the involved activities and artifacts.

The goal of our process is automating the Application Engineering stage. Particularly, once the functionality of a new product is defined by a Feature Configuration Model, the Product Architecture is automatically derived from it, and the product implementation supporting such an architecture can also be automatically generated. To achieve such a goal, we apply Model-Driven Engineering [27] techniques to define a systematic process to perform the Domain Engineering stage. We conceive all participating artifacts as models, rendering rigorous and unambiguous artifacts amenable to be manipulated by tools. A metamodeling approach is used to define the domain-specific languages needed to express each participating artifact. Altogether, the automation of the Application Engineering stage is achieved by means of model transformations that are built during Domain Design, and applied during the Application Engineering stage. To achieve this goal, we define our process based on the following six main ideas.

(1) **Features represent functionality.** There are different proposals for documenting domain models including both functional and quality requirements, such as [25]. In most of
them, commonalities and variabilities are captured by means of a feature model. We constrain feature models to include features representing functionality, services, parameters or data storages, and we assume quality requirements to be documented in a separated artifact. Also we assume that feature models to express variability only at the functional level, and not at the quality attribute level. We propose the architect to organize the architectural decisions made during Domain Design in terms of the features in the Feature Model, however we also propose he/she to take into account this quality requirement artifact and existing implementation assets when making such decisions.

(2) Features lead component architecture construction. Domain Design traditionally focuses on the construction of the PLA that embodies the critical design decisions that address functionality and quality, and also commonality and variability of all products in the SPL. In our approach, we organize these decisions in terms of the features in the Feature Model, which in turn, guide the compositional structure of the architectural components. Each feature that may be selected as part of a product inspires a set of architectural decisions that guides the construction of part of the architecture of a product that includes that feature. Decisions are made locally to each particular feature, only considering its direct member features. Following such an approach, quality-related decisions are associated with features near the root of the Feature Model, while functional-specific decisions are associated with features near the leaves. We do not address additional constraints like require and exclude, or other relationships like composed-of, generalization/specialization, and implemented-by among features in this work. As this issue is both relevant and complex, it is currently part of our ongoing work.

(3) Record the architecting activity, not the architecture. In the traditional approach, the Domain Design develops the PLA, usually yielding complex architecture definitions in non-standard architecture description languages. During Product Design, all variabilities in the PLA are resolved to obtain a particular Product Architecture (PA). While Product Analysis resolves variability at the feature level, Product Design resolves variability at the architectural level; then, the effort is somehow duplicated. Having no direct traceability from features to architectural components, and mainly to architectural decisions, hardens tool-assistance in the construction of a PA. Besides, such an approach lacks first-class representation for design decisions. Although they are implicit in the resulting architecture artifact, in the traditional approach the underlying rationale is scattered among the participating components and the general structure. In our approach, we record the product line architecting activity instead of the PLA. For each feature in the Feature Model, we preserve the set of decisions involved in providing this feature by the architecture. Such decisions are explicitly recorded as the set of actions that must be performed on a PA to support the feature. These actions are described in terms of model transformation rules that output a fragment of the PA model when the particular feature is present in the product. Then, the whole set of model transformation rules corresponding to the features selected during the product analysis constitutes the core of the model transformation that produces a particular PA from the Feature Configuration Model. Different transformation technologies have different characteristics such as rule scheduling, read/write-only access to source and target models, among others [9]. These characteristics need to be taken into account while developing each independent transformation.

(4) Components lead implementation. In the traditional approach, the Domain Implementation develops, refactors or buys the component implementations that participate in particular product implementations. Besides, a generator program is usually built so as to integrate these components. In our approach, we modularize such a generator in terms of the components in the PAs. There are two kinds of components: those not further decomposed and whose component implementations are developed, and those further decomposed, and hence, the architectural description defines how they are designed in terms of the inner components. For each composed component, a component generator is developed during Domain Implementation which assembles it, considering the variations in its internal composition as described by the transformation rules defining the component. Then, there is traceability from architectural components to component implementations or component generators. The integration of all component generators can be regarded as the general generator of the traditional approach. Such an integration conforms the model transformation that obtains a product implementation for a particular Product Architecture.

(5) Incrementally develop the product line. In our process, the defined Domain Engineering artifacts can be built incrementally. While a complete Feature Model is usually built during Domain Analysis, the other artifacts can be produced incrementally by addressing only those features that are required by each particular product under development. Our modularization strategy not only favors incrementality, but also evolvability as changes in the SPL scope have restricted impact on other developed artifacts. The development effort is greater for the first products as the top-most features and most of the composed components will probably participate in all products, and hence such features and components need to be tackled early in the process.

(6) Abstract underlying technology. As the defined approach relies on Model-Driven Engineering (MDE), particular technologies need to be used for constructing our artifacts. For instance, a particular model transformation language is required to define the set of rules that produces the Product
Architecture. Also, a particular language is required for coding component generators. To achieve evolvability, we abstract away the underlying technology in the metamodels, enabling the seamless integration of new/different MDE technologies.

3. Domain Engineering Stage

Domain Analysis. Feature Models have shown to be useful and widely-used for documenting domain analysis [3][8]. Thus, the goal of this activity is to produce a Feature Model such that:

- The leaf features include those that can be encapsulated in a single coherent functional unit. Thus, the leaf nodes of the model must represent specific functionality provided by a product, parameterization of such a functionality, user interaction, or access to data storage.
- The internal features (i.e. those with subfeatures) represent functional areas of the SPL that can be provided by means of the interaction or combination of the functionality provided by the features they depend on, i.e. their children features.

The metamodel we use for building Feature Models is a simplification of the metamodel proposed by Czarnecki et al. [10]; we depict it in Figure 2. All Features in the Feature Model have distinct names and may have composing members. Root features are used to modularize the model; they cannot be members of other features, and exactly one of them must be marked as main in the model. Solitary and Grouped features represent those that are ungrouped and grouped, respectively. Members of composed features can be Solitary, Reference to a particular Root feature, or Group. A Group consists of a set of Grouped or Reference features. Variability is represented by the cardinality: for Solitary features, cardinality indicates how many times it can be used to compose the owner feature, and for Groups, cardinality indicates how many members can be actually used.

Domain Design. The goal of Domain Design is to make the critical decisions on the structure of the product architecture and on the resolution of quality attributes. Architectural patterns are used in order to address the quality and functional requirements which are documented in requirement specification artifacts. The Feature Model is used to organize the decision making activity during Domain Design. Provided that features in the Feature Model represent functional aspects, we follow the tree-structure of such a model to modularize the architectural decisions. Our approach is centered in explicitly recording the architecting activity, not simply the architectural products. The goal of Domain Design is to record for each feature the architectural decisions that are made to address the functionality and variability represented by such a feature in the architecture. Quality attributes and existing implementation assets are also considered, mainly when recording design decisions associated to those features near the root of the Feature Model.

As we will explain in Section 4, we understand Product Design as a model transformation from a Feature Configuration Model to a Product Architecture. Thus, the architectural decisions made during Domain Design are recorded as fragments of this model transformation. Each fragment consists of a set of rules encapsulating the knowledge of how to build the Product Architecture when the corresponding feature is present in the Feature Configuration Model. The Product Architecture is organized in terms of a single architectural view based on the Component & Connector viewtype [5]. Then, the rules populate such an artifact with components whose provided and required interfaces are assembled by connectors. Leaf features probably yield component interfaces or components that are not further decomposed. Internal features yield components that are further decomposed in terms of interconnected subcomponents which correspond to some of their subfeatures.
Domain Design produces a Feature-to-Architecture Transformation Rule artifact, expressed in terms of the metamodel in Figure 3. A PLA element is formed by a set of declarations and a top feature. Each declaration corresponds to a general declaration that can be used by the rules attached to each feature. Features have distinct names, and are organized in a tree-structure inspired by the Feature Model. The name of the Feature is used for matching purposes with the features in an input Feature Configuration Model. Each Feature has a set of rules to indicate how to affect an output Product Architecture when the given feature is present in a Feature Configuration Model. Declaration and Rule metaclasses are abstract for portability purposes. Specializations of the metamodel can be made, targeting different model transformation technologies. In Figure 3, we also illustrate one of such specializations targeting the AtlanMod Transformation Language (ATL) [17] even though other transformation languages could have been used as well. An ATLDeclaration can include either a CALLEDRULE or a HELPERS, both metaclasses of the ATL metamodel. A particular ATLRule consists of: (i) a filter OCLExpression to distinguish among different cases of the input feature (e.g. whether a particular child feature is present or not), (ii) various RULE_VARIABLE_DECLARATIONS for rule-specific constants, (iii) an OUT_PATTERN indicating the elements in the target Product Architecture model that must be present, and (iv) an ACTION_BLOCK for imperative actions for the rule. These metaclasses are defined in the ATL metamodel and they conform the core composing elements of a general ATL rule in such a metamodel.

Domain Implementation. The goal of Domain Implementation is to build, refactor or buy the component implementations for all the logical components participating in the architecture of the products of the SPL. We organize the product line implementation (PLI) as a model expressed in terms of the metamodel depicted in Figure 4.

A PLI is modularized in terms of Components, inspired by the logical components resulting from the rules in Feature-to-Architecture Transformation Rule. There are two kinds of Components: TERMINAL or COMPOSED. A TERMINAL component is not further decomposed in the architecture, and only its interfaces are specified. For each component of this kind, one or more IMPLEMENTATIONS must be made available, either by implementing or buying them. A COMPOSED component is further decomposed into interconnected subcomponents. For each component of this kind, the composing member components are preserved, and one or more GENERATORS must be developed. To this end, a script, program or transformation is developed which is able to generate the component implementation for the corresponding component. A GENERATOR encapsulates the knowledge of how to implement a COMPOSED component, joining the implementations of the member components and generating any necessary glue code. For both kinds of components, IMPLEMENTATIONS and GENERATORS can be targeted to different particular platforms, which is annotated in their deployment property. In particular, in the metamodel in Figure 4 we include the specialization targeting the Java platform for which both JAVAIMPLEMENTATIONS and JAVAGENERATORS are coded in a JAVA ARCHIVE.

4. Application Engineering

Product Analysis. The goal of Product Analysis is the selection of the desired features for a particular product. These features are selected from those provided by the SPL, considering variability constraints. Thus, a Feature Configuration Model defines which configuration of the Feature Model represents the product to be developed and consists of Features composed by subfeatures. A Feature Configuration Model is an instance of the metamodel in Figure 5; the Feature Model constrains which Feature Configuration Models can be actually defined.

Product Design. The goal of Product Design is to define the Product Architecture for the particular product being developed, considering its desired features defined in the Feature Configuration Model. The architectural
The feature included in the product under development is a subset of the model transformation rules corresponding to the product domain design decisions made during Domain Design and/or depending on implementation aspects of the Application Generator. Provided our ATL specialization, we successfully implemented meta-transformations that transform a feature targeting a given MDE technology, and an associated meta-transformation that transforms a feature targeting an ATL configuration into an ATL transformation. Although to define the meta-transformation requires considerable effort, once developed it can be reused in any SPL development project that uses the same MDE technology. Provided our ATL specialization, we successfully implemented the corresponding meta-transformation that transforms a feature targeting an ATL transformation. This derived transformation is then applied to the Feature Configuration Model to obtain the particular Product Architecture. By this means, the Product Design activity is fully automated. The resulting architecture is an instance of the metamodel in Figure 6. This metamodel is a simplification of the Composite Structure metapackage of the UML 2.11 Superstructure Specification [29].

5. Addressing the Meshing Tool Domain

Meshes are used for numerical modeling, visualizing and/or simulating objects or phenomena. A mesh is a discretization of a certain domain geometry that can be either composed by a unique type of element, such as triangles, tetrahedra or hexahedra, or by a combination of different types of elements. Meshing tools generate and manage these discretizations. Such tools are inherently sophisticated software due to the complexity of the concepts involved, the large number of interacting elements they manage, and the application domains where they are used. Provided that meshing tools are used in a variety of different application domains, they may require slightly different functionalities.

As these tools have usually been developed with ad hoc methodologies, and without taking reuse as a goal, every new tool needs to be developed from scratch even though it may involve algorithms already implemented and data structures already designed, all of them also used and tested. Meshing tools have a good opportunity for reuse, and hence their development using SPL is promising.

We use the example of the development of a Meshing Tool SPL [13][23] for illustrating our approach. In this section we briefly overview the key aspects of the analysis and design activities of both Domain Engineering and Application Engineering stages. In order to make our experiment repeatable, we provide a complete guide which thoroughly describes the involved activities, roles, artifacts and tools in http://mate.dcc.uchile.cl/research/tools/mddofspl/. The case study was developed using the ATL Bundle 2.0 UML2 Version for Windows consisting of Eclipse 3.3.0 with the ATL plug-in pre-installed. Also, the FeaturePlugin r0.6.6 and OrangevoltXSLT 1.0.7 plug-ins are required. We defined all metamodels using KM3 so as to generate the corresponding ECore version. Also, text-to-model and model-to-text transformations were implemented in XSLT, and model-to-model and meta-transformations were coded in ATL.
5.1. Domain Engineering.

During Domain Analysis we use the FeaturePlugin to define the Feature Model; we illustrate this artifact in Figure 7, which describes six functional areas involved in a Meshing Tool. The User Interface feature represents all possible user interfaces for a product. Geometry indicates different mechanisms to load into the tool a representation of the object to be modeled, in different input formats. Generate Initial Mesh provides several algorithms for transforming an input geometry to a Mesh. These algorithms generate either 2D or 3D meshes. The initial mesh could need to be changed, both in quantity and size of its elements; here we can use different Algorithms. Finally, the mesh can be saved in different Output Formats. We developed a text-to-model transformation which transforms the XML file produced by the FeaturePlugin to the corresponding model instance of the metamodel in Figure 2.

During Domain Design, the Feature-to-Architecture Transformation Rule artifact is built. First, we use a model-to-model transformation we developed to create an initial version of this model from the Feature Model, only containing all defined features and their member relationship. Second, the model is manually augmented to include the required declarations, together with the rules for each feature. We present in Figure 8 a fragment of one of the rules using the ATL specialization of our metamodel illustrated in Figure 3, using textual notation.

The rule corresponds to the Meshing Tool feature (line 1) in the case where the optional Algorithms feature is selected (line 2); f represents the Feature element of the source Feature Configuration Model. In this rule we encode the decision of which architectural patterns govern the overall structure of the product architectures; in our case study we apply a hybrid architectural style, based on the 3-tier pattern where the two bottom-most tiers follow the shared-data pattern [4]. The rule requires a component c to be present in the target Product Architecture model (lines 6–10), with the same name as the feature and whose subcomponents are those generated by the rules corresponding to the subfeatures of f (line 7). The connectors for c are those defined in this rule. Two examples are included in the figure: a connector linking the User Interface and Geometry subcomponents (lines 11–16), and several connectors linking the User Interface to each provided interface of the Algorithms component (lines 17–23).

5.2. Application Engineering.

During Product Analysis we use the FeaturePlugin to create the Feature Configuration Model defining the desired features in the new product being built; Figure 9 illustrates the selected features. We use a text-to-model transformation to obtain this model as an instance of the metamodel shown in Figure 5.

During Product Design, the meta-transformation is used to generate from the Feature-to-Architecture Transformation Rule the Feature-to-Architecture Transformation artifact. This transformation is then applied to the Feature Configuration Model to automatically generate...
Product Architecture

Figure 9. Feature Configuration Model for a Meshing Tool.

Figure 10. Product Architecture - fragment for Meshing Tool feature.

the PRODUCT ARCHITECTURE. Figure 10 illustrates a fragment of the resulting PRODUCT ARCHITECTURE model generated by the rule shown in Figure 8, applied to the FEATURE CONFIGURATION MODEL shown in Figure 9. The MESHING TOOL component is composed by the subcomponents generated by the rules attached to the subfeatures of the MESHING TOOL feature.

6. Related Work

Matinlassi [22] shows a technique, namely QAMT, for transforming architecture models to new models according to defined requirements, in the context of SPL; these transformations are from one architecture model to another architecture model, in other words, a refactoring of an architecture model. This approach does not document explicitly which requirements are changing, and therefore this knowledge can only be held by the architect. In our approach we only consider quality requirements for the Domain Design, but we focus on functional requirements by changing the feature diagram and then adding new rules or changing the existing ones in the FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE set. Although QAMT is not an automatic tool, the author shows that it is a viable possibility. Finally, the author makes use of the available knowledge about architectural and design patterns, and also model transformations. We consider the same knowledge, but we also define specific rules for producing concrete products.

Liu and Mei [20] present an approach for mapping requirements stated as feature models to software architecture, looking for traceability and consistency. They include both functional and non-functional features. We only consider functional features as part of the feature model, while quality attributes should be recorded in another requirement artifact, and used by the architect when building the FEATURE-TO-ARCHITECTURE TRANSFORMATION RULES. In [20] features are mapped to the conceptual, logical and deployment architectural views. In our process the architect can only choose an architectural style in the C&C viewtype although this is not an intrinsical limitation. Liu and Mei do not establish a mapping between requirements and architecture in the different views, but they show the real possibility of doing so. Also, the authors do not deal with variation points.

Savolainen et al. [26] also map requirements, features and architectural assets. However, they locate features in the solution domain instead of the problem domain, thinking of features with an implementation perspective. This work is similar to ours in two ways: our architectural assets are located in the leaves of the feature model, and mapping rules are explicitly designed for features in any level in the feature model (internal or leaf features).

Laguna et al. [19] focus on traceability between features and architectural models based on UML. Their approach associates each feature to a package containing classes and relationships, thus, some features are transformed into classes and others into packages. Their transformations are defined in QVT similarly to ours that are defined in ATL. Besides, both methods preserve rationale using a slightly different feature meta-model. Their approach deals with transformations at the class level while ours deals with the component level. We also deal with the automation of product implementation.

In [15] features are considered as transformations that modify programs whenever they are included in the product under development. Their approach associates each feature to a package containing classes and relationships, thus, some features are transformed into classes and others into packages. Their transformations are defined in QVT similarly to ours that are defined in ATL. Besides, both methods preserve rationale using a slightly different feature meta-model. Their approach deals with transformations at the class level while ours deals with the component level. We also deal with the automation of product implementation.

Czarnecki et. al. [7] propose a template-based target-oriented approach for generating models from feature models. They deal with general models, but their approach can be also applied to architectural models. We address the construction of product architectures applying similar
techniques, but our transformations follow a source-based hybrid approach. Moreover, we are not concerned with the generation of general models but mainly with a more concrete and complex problem which is the generation of a PLA and eventually the implementation.

7. Conclusions & Further Work

In this work we applied Model-Driven Engineering techniques to define a Software Product Line development process that systematizes the Domain Engineering stage so that Application Engineering is automated. Our experience applying the process to the Meshing Tool domain was successful in building product architectures. However, implementation was only addressed at the process level because only some of the component implementations were available and not all of them satisfied the required interfaces. Systematically building the software architecture allows us to achieve consistency between requirements specified in the form of the feature model [3][29]. Also the sequence of model transformations applied in the design process enables traceability, another desirable quality. The scalability of the approach was not part of the goals of this paper. However, as it is an important quality, also suggested by Berg et al., we plan to tackle it in further work.

In the traditional approach to SPL, the PLA is designed considering the complete feature model, so changes in a feature affect the whole design. In our approach we make architectural decisions only considering the information about the children features, so changes in a feature have a local impact. However, we realize that it may be useful to also consider some information about siblings or a complete subtree in order to make better decisions; e.g., in the case study it was necessary to know which ALGORITHMS were chosen in order to select the right USER INTERFACE features. However, if much is considered, we may end up following the traditional approach.

SPL approaches are centered in architecture, so it is highly recommended to assess the PLA. Our approach does not provide an explicit PLA, it is instead implicitly defined by the FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE artifact. Therefore, our PLA cannot be assessed with traditional methods. Our process generates explicit product architectures that can be assessed with traditional methods [12], but it could be expensive if the number of products in the SPL is large. Nevertheless, representative product architectures can be automatically generated so as to perform architecture assessment.

Our approach to software architecture definition is independent of the particular architectural representation that is decided for organizing the PRODUCT ARCHITECTURE. Although we currently use a single architectural view based on the C&C viewtype for representing the PA, any other representation could be used. We only need to redefine our PRODUCT ARCHITECTURE METAMODEL to include other viewtypes. However, the more complex the representation, the greater the effort to define the set of rules and to preserve consistency among them will be. We have already experienced developing several coordinated views and a prototype supporting tool has also been developed proving feasibility of the extended approach [24].

Designing and building each transformation rule associated to each feature is definitely the most complex part of the process, and as any architectural design activity, requires high qualification. However, this complexity is faced with a divide-and-conquer approach, so each transformation can be incrementally built and improved, and once developed the knowledge contained can be shared and reused.

Domain implementers may generate domain-specific languages (DSLs) for product definition in order to assist building particular SPL members by defining their syntax and building the supporting tools. Using our approach, it would only be necessary to define the DSL syntax and the transformation from expressions in such a language to a FEATURE MODEL CONFIGURATION. Then, the generation of the product is achieved following the automation provided by our approach.

Although testing is of main importance in the context of product lines due to high reuse, in this paper we do not cover testing activities. Core implementation asset, the product-specific software, the interactions between them, and the products should be tested [6].

Currently we specify quality attributes in separated artifacts, even though there are some authors that extend feature models to also consider quality attributes [2][14]. We recognize the need to incorporate their systematic management and we are considering it as part of our current work. Also, it would be desirable to count on an integrated tool support for the complete process.

Acknowledgments

The work of Daniel Perovich has been partly funded by CONICYT Chile. The work of Pedro O. Rossel was partially supported by grant No. UCH 0109 from MECESUP, Chile and by grant of the Departamento de Postgrado y Postítulo de la Vicerrectoría de Asuntos Académicos de Universidad de Chile.

References


