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

Pedro O. Rossel, Daniel Perovich, and María Cecilia Bastarrica

Department of Computer Science, Universidad de Chile
{prossel,dperovic,cecilia}@dcc.uchile.cl

Abstract. A Software Product Line (SPL) is a portfolio of similar software products that target a particular domain. SPL methodologies generally use Feature Modeling to express requirements including variability, and provide a prescribed way to develop particular products from reusable assets. These methodologies do not explicitly preserve design rationale, which is implicitly stated in the SPL architecture. Having a systematic, tool-enabler, scalable and evolvable method for generating family members is desirable. In this paper, we use Feature Configuration Models (FCM) as the DSL for specifying particular product requirements, and we apply MDE techniques for systematizing the process of product generation. We use model transformations for stating how the Product Architecture is built from the FCM, and for integrating the reusable components. Such transformations share a common but evolvable set of rules, and conform an explicit representation of the SPL rationale. We apply our approach for developing a Meshing Tool SPL.

1 Introduction

Software Product Lines (SPL) is an approach to develop related systems reusing a managed set of core assets sharing functionality and quality attributes [4]. Most SPL development processes identify three stages. In the *Domain Engineering* (DE) stage reusable assets are developed and maintained, and the scope and production plan are defined. In *Application Engineering* (AE) particular product requirements are gathered, and the product is built by arranging the reusable assets according to the production plan. If some product requirements fall beyond the scope the *Management* stage determines whether the product will be built or not. The DE stage is formed by three activities: Domain Analysis, Domain Design, and Domain Implementation [18]. The Domain Analysis produces the Domain Model. 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 Domain Model gathers all the potentially required features for any particular product in 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 AE 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.

Traditional approaches to SPLs are well defined. However, the design decisions that must be made, such as designing the PLA, require to take into account the whole information the domain analysis may produce. This situation puts stress on the domain analysis activity and the artifacts it produces. Feature models are a widely used approach to domain analysis capturing commonalities and variabilities [6], and there are several methods and notation for generating them [1, 11]. 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 [7]. There have been some successful experiences in automating product generation in SPLs [9, 14], most of them targeting specific domains. Automation is thus desirable and also feasible. Tools supporting product generation should be scalable, traceable, and manage consistency and visualization for variability among different artifacts [2].

We define a SPL development process based on the Praise reference process [18]. We apply MDE [17] techniques to provide a systematization of the DE stage, which enables the automation of the AE stage. To this end, we consider features in the Feature Model to represent main functional areas. Architectural decisions are explicitly recorded as model transformation rules attached to each feature. For each component either an implementation or a generator is implemented. Therefore, generating a particular product consists of defining a particular Feature Configuration Model only. 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 generators of those components participating in the product architecture.

The rest of the paper is structured as follows. Section 2 presents the defined development process, including first its rationale, and then detailing the activities and artifacts involved in both, the DE and the AE stages. We successfully applied this process by addressing the Meshing Tool domain[1], and report a key part of this experience in Section 3. Related work is reviewed in Section 4. Finally, Section 5 concludes with a discussion and suggests further work directions.

2 Model-Driven Development of Software Product Lines

In this work, we define a process based on the Praise reference process [18] for SPLs. We only deal with *Domain Engineering* and *Application Engineering*,

[1] `http://mate.dcc.uchile.cl/research/tools/mddofspl/`

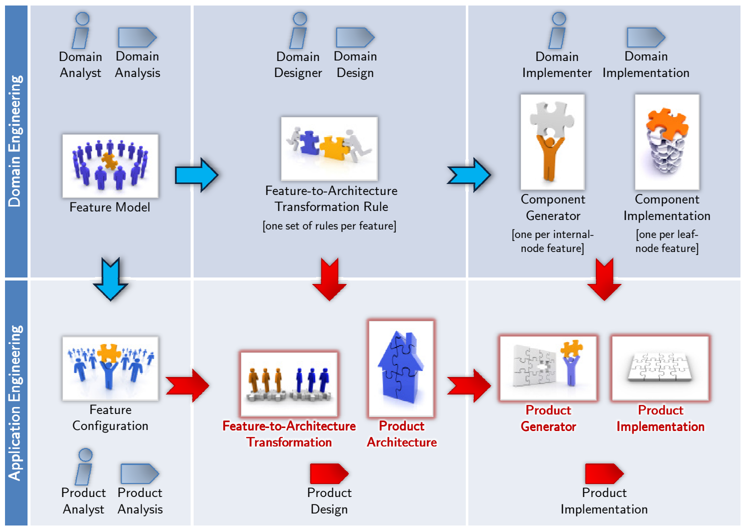

Fig. 1. Model-Driven Development Process of Software Product Lines.

both organized in terms of three major activities. First, *Analysis* uses features to explicitly define functionality, and also variability in the case of DE. Second, *Design* is architecture-centric, and tackles critical structural and quality-addressing decisions. Third, *Implementation* deals with actual component and product development. Figure 1 illustrates the proposed process.

The major goal of our process is automating AE. Particularly, once the functionality of the new product is defined in the 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 define a systematic process to perform DE and we apply MDE [17] techniques. 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 DSLs needed to express each participating artifact. Altogether, the automation of AE is achieved by means of model transformations that are automatically derived from the models defined in DE, and applied during AE.

2.1 Process Rationale

Features represent functionality. We constrain Feature Models only to features representing services, functionalities, parameters or data storages. Quality attributes need to be documented in a separated artifact as they are needed for design. However, as proposed in [13], a Feature Model can be refined into one that only refers to functional capabilities identifying quality attributes also as features.

Features lead component architecture construction. Domain Design focuses on the construction of the PLA that embodies the critical design decisions that address functionality and quality, and also commonality and variability. 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 inspires an architectural component that encapsulates the set of decisions that guides the component internal organization. Decisions are made locally to each particular feature, only considering its direct member features. 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.

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 ADLs. 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, 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 include the feature support. This 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 are the core of the model transformation that produces a particular PA from the product capabilities description, namely the Feature Configuration Model.

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 automate this task. 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 other components. For each compound component, the Domain Implementation develops a component generator which assembles it, considering the variations in its internal composition as described by the transformation rules. 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 from a particular PA.

Incrementally develop the product line. In our process the defined DE 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. The modularization strategy not only favors incrementality, but also scalability as changes in the SPL scope have restricted impact on other developed artifacts. The development effort would be greater for the first products as the top-most features and most of the compound components will probably participate in all products and hence, need to be tackled early in the process.

Abstract underlying technology. As the defined approach relies on 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 PA. 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 technologies.

2.2 Development Process Activities and Artifacts

As illustrated in Figure 1, our process involves the *Domain Engineering* and *Application Engineering*, and organizes each of them in three activities: *Analysis*, *Design* and *Implementation.* Although these activities are sequentially presented, they are strongly related and they can be performed incrementally. First, we describe DE with its activities and core assets. Then we describe how in AE the desired product is selected and its design and implementation is automatically built using the core assets of DE.

Domain Engineering

Domain Analysis. Feature Models have shown to be useful and widely-used for documenting domain analysis [2, 5]. 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 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 include those with subfeatures. Thus, the internal nodes of the model must 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. [6]; 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.

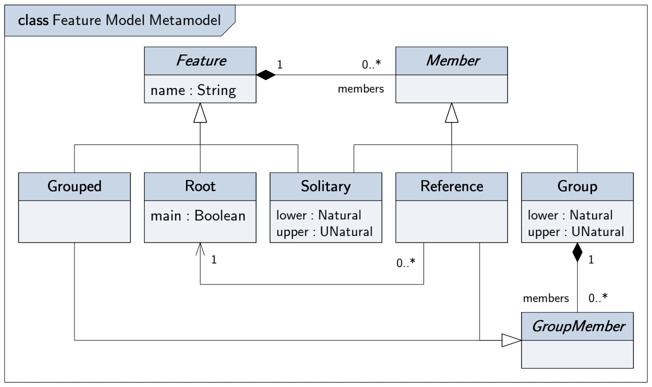

Fig. 2. Feature Model Metamodel.

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 group 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. 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 product architectural structure and 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 in the 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 are also considered, mainly when recording design decisions associated to those features near the root of the Feature Model.

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 [3]. 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

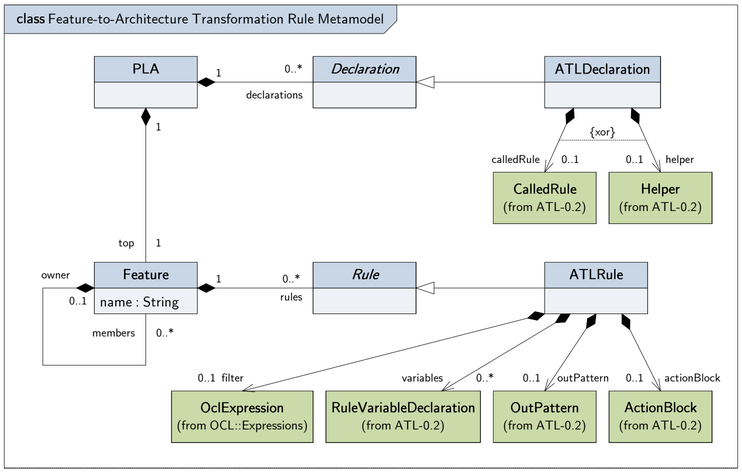
class Feature-to-Architecture Transformation Rule Metamodel
PLA
Declaration
ATLDeclaration
declarations
1
0..*
{xor}
calledRule
0..1
0..1
helper
CalledRule (from ATL-0.2)
Helper (from ATL-0.2)
top
1
Feature
name : String
owner
0..1
members
0..*
Rule
rules
ATLRule
0..1 filter
variables
0..*
0..1 outPattern
0..1 actionBlock
OclExpression (from OCL::Expressions)
RuleVariableDeclaration (from ATL-0.2)
OutPattern (from ATL-0.2)
ActionBlock (from ATL-0.2)

Fig. 3. Feature-to-Architecture Transformation Rule Metamodel.

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 an input Feature Configuration Model. Declaration and Rule metaclasses are abstract for evolvability 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 Atlas Transformation Language (ATL). An ATLDeclaration can include either a CalledRule or a Helper, 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 RuleVariableDeclarations for rule-specific constants, *(iii)* an OutPattern indicating the elements in the target Product Architecture model that must be present, and *(iv)* an ActionBlock 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 develop the components participating in the architecture of the SPL products. The product line implementation (PLI) is organized as a model expressed in terms of the

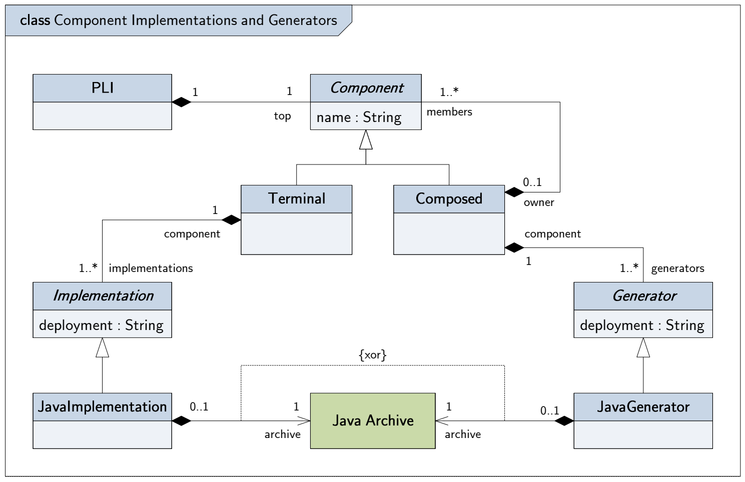

Fig. 4. Component Implementations and Generators Metamodel.

metamodel in Figure 4. It 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. *(i)* A Terminal component is not further decomposed in the architecture, and for which only its interfaces are specified. For each component of this kind, one or more Implementations must be available. *(ii)* 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 members components and generating any necessary glue code. For both cases *(i)* and *(ii)*, Implementations and Generators can be targeted to any particular platform, 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.

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. Feature Configura-

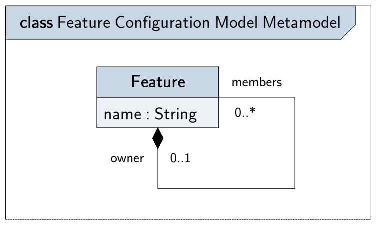

Fig. 5. Feature Configuration Model Metamodel.

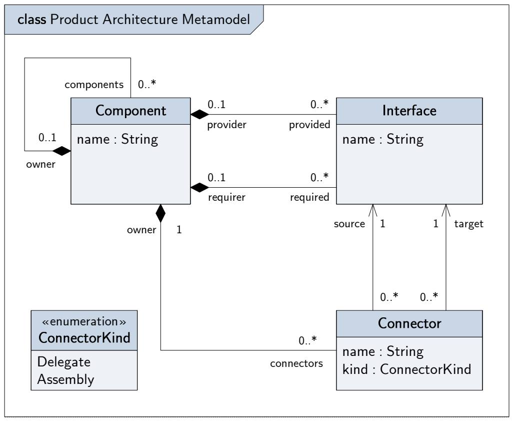

Fig. 6. Product Architecture Metamodel.

tion Model is an instance of the metamodel in Figure 5; it is the Feature Model that constrains which Feature Configuration Models can be actually developed.

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 decisions made during *Domain Design* must be used to produce the Product Architecture; the subset of transformation rules corresponding to the features included in the product under development are used to derive the architecture.

To this end, a meta-transformation is defined which takes a particular Feature-to-Architecture Transformation Rule artifact targeting a given technology, and produces a Feature-to-Architecture Transformation for that transformation technology. One meta-transformation is required for each technology used to specialize Feature-to-Architecture Transformation Rule. However, once developed, this meta-transformation can be reused in any SPL development project. Provided our ATL specialization, we implement the corresponding meta-transformation that transforms a Feature-to-Architecture Transformation Rule artifact to 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.

Product Implementation. The goal of Product Implementation is to build the actual product, considering the architectural organization defined in the Product Architecture. Once decided a particular target technology, the corresponding Component Implementations and Generators developed during the *Domain Implementation* must be used to obtain the implementation of the product.

To this end, a Product Generator that is capable of generating the Product Implementation must be built. Proceeding analogously to the *Product Design*, the Product Generator can be automatically derived from the set of Component Generators developed by defining a meta-transformation that uses them according to which component implementations need to be generated. Then, such a Product Generator takes the particular Product Architecture and generates all the component implementations of the composed components, which, in turn, rely on the Component Implementations for the terminal components. Provided this meta-transformation and the component implementations for the terminal components, the Product Implementation activity is fully automated.

3 Addressing the Meshing Tool Domain

We applied the process to the development of a Meshing Tool SPL [15]. In this section we briefly overview the key aspects of the analysis and design activities of both DE and AE. In order to make our experiment repeatable, we provide a complete guide which thoroughly describes the involved activities, 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.

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.

3.1 Domain Engineering.

During *Domain Analysis* we use the FeaturePlugin to define the Feature Model; we illustrate this artifact in Figure 7, which describes the 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

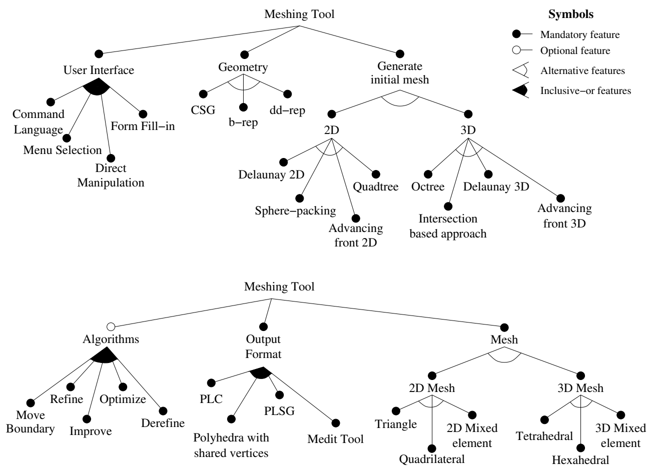

Fig. 7. Feature Model for Meshing Tools.

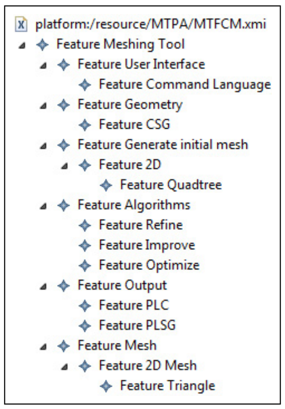
```
platform:/resource/MTPA/MTFCM.xmi
  Feature Meshing Tool
    Feature User Interface
      Feature Command Language
    Feature Geometry
      Feature CSG
    Feature Generate initial mesh
      Feature 2D
        Feature Quadtree
    Feature Algorithms
      Feature Refine
      Feature Improve
      Feature Optimize
    Feature Output
      Feature PLC
      Feature PLSG
    Feature Mesh
      Feature 2D Mesh
        Feature Triangle
```

Fig. 8. Feature Configuration Model for a Meshing Tool.

input geometry to a Mesh. These algorithms generate 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 9 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. The rule requires a component c to be present in the target Product Architecture model (line 5), with the same name as the feature and whose subcomponents are those generated by the rules corresponding to the subfeatures of f (line 6). 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 9-13), and several connectors linking the User Interface to each provided interface of the Algorithms component (lines 14-18).

```
ATLRule for 'Meshing Tool' {
  filter f.members→select(fi |fi.name = 'Algorithms')→notEmpty();
  variable inames : Sequence(String) = thisModule.getAlgorithmFeatures(f)→collect(fa |fa.name)→asSequence();
  out {
    c : PAMM!Component (
      name ←f.name, components ←f.members,
      connectors ←Set{xgeometry, xgenerate, xgeneratemesh, xoutput, xoutputmesh, xalg, xalgmesh},
    ),
    xgeometry : PAMM!Connector (
      name ←'Geometry', kind ←#Assembly,
      source ←c.components→any(ci |ci.name = 'User Interface').required→any(ii |ii.name = 'IGeometry'),
      target ←c.components→any(ci |ci.name = 'Geometry').provided→first()
    ),
    xalg : distinct PAMM!Connector foreach(iname in inames) (
      name ←iname, kind ←#Assembly,
      source ←c.components→any(ci |ci.name = 'User Interface').required→any(ii |ii.name = 'I' + iname),
      target ←c.components→any(ci |ci.name = 'Algorithms').provided→any(ii |ii.name = 'I' + iname)
    ),
    ...
  }
}
```

Fig. 9. Feature-to-Architecture Transformation Rule for the Meshing Tool feature.

3.2 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 8 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 the Product Architecture. Figure 10 illustrates a fragment of the resulting Product Architecture model generated by the rule shown in Figure 9. The Meshing Tool component is organized by the subcomponents generated by the rules attached to the subfeatures of the Meshing Tool feature. Such an organization follows a hybrid architectural style, based on the 3-tier pattern where the two bottom-most tiers follow the shared-data pattern.

4 Related Work

Several authors have proposed approaches to relate feature models and product architectures. A good review about approaches for overcoming this gap is found in [10]. For Berg et al. [2], approaches for variability management relating requirements and solutions should be consistent, scalable, traceable an provide visualization media. They state that, 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 non-scalable and not traceable either. Visualization is the only aspect fulfilled by feature models. In our approach, the incremental construction of the product architecture enhances scalability and the use of Feature-to-Architecture Transformation Rules makes it also traceable.

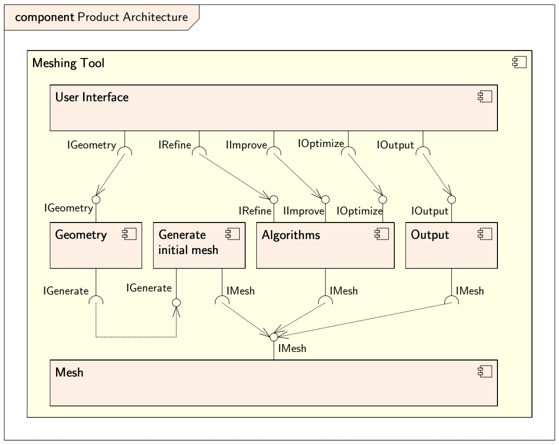

Fig. 10. Product Architecture - fragment for Meshing Tool feature.

Liu and Mei [13] 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 [13] 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 it. Also, the authors do not deal with variation points.

Savolainen et al. [16] 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. [12] 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 as we defined ours in ATL. Besides, both methods preserve rationale using a slightly different feature meta-model. One of the strengths of this work is the transformation to UML models. Their approach deals with transformations at the class level while ours deals with the component level.

In [9] features are considered as transformations that modify programs whenever they are included in the product under development. Their approach is similar to ours, but they skip the product architecture and focus on product implementation directly.

5 Conclusions & Further Work

In this work we applied MDE techniques to define a SPL development process that systematizes Domain Engineering 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.

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 SPL methods. Our process generates explicit product architectures that can be assessed with traditional methods [8], but it could be expensive if the number of products in the SPL is large. This design approach 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 others viewtypes. However, the more complex the representation, the greater the effort to define the set of rules and to preserve consistency will be.

Domain implementation may generate DSLs to aid building particular SPL members by defining its syntax and building the supporting tools. With our approach it is necessary to design the DSL syntax, but not to implement any tool. We only implement transformations from the DSL metamodel to the Feature Configuration Model Metamodel, taking advantage of our process and infrastructure.

Currently quality attributes are specified in separated artifacts. We recognize the need to systematically incorporate their management and we suggest it as further work. Also, it would be desirable to count on an integrated tool support for the complete process.

References

1. T. Asikainen, T. Männistö, and T. Soininen. A Unified Conceptual Foundation for Feature Modelling. In *10th International Software Product Line Conference (SPLC 2006)*, pages 31–40, 2006.
2. K. Berg, J. Bishop, and D. Muthig. Tracing Software Product Line Variability: From Problem to Solution Space. In *Conference of the South African Institute of Computer Scientists and information technologists on IT research in developing countries (SAICSIT '05)*, pages 182–191, 2005.
3. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures. Views and Beyond.* SEI Series in Software Engineering. Addison-Wesley, 2002.
4. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, 2001.
5. K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications.* Addison Wesley, 2000.
6. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Third International Software Product Lines Conference Software Product Lines (SPLC 2004)*, volume 3154 of *LNCS*, pages 266–283, 2004.
7. D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying Software Product-Line Architecture. *IEEE Computer*, 30(8):49–55, 1997.
8. L. Dobrica and E. Niemelä. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.
9. G. Freeman, D. Batory, and G. Lavender. Lifting Transformational Models of Product Lines: Case Study. In *1st International Conference on Model Transformations, (ICMT'2008)*, Zurich, Switzerland, 2008.
10. M. Galster, A. Eberlein, and M. Moussavi. Transition from Requirements to Architecture: A Review and Future Perspective. In *Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006)*, pages 9–16, 2006.
11. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Nov. 1990.
12. M. A. Laguna, B. González-Baixauli, and J. M. Marqués. Seamless Development of Software Product Lines. In *6th International Conference on Generative Programming and Component Engineering (GPCE '07)*, pages 85–94, 2007.
13. D. Liu and H. Mei. Mapping requirements to software architecture by feature-orientation. In *Second International SofTware Requirements to Architectures Workshop (STRAW'03)*, pages 69–76, 2003.
14. F. Losilla, C. Vicente-Chicote, B. Álvarez, A. Iborra, and P. Sánchez. Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. In *ECSA 2007*, volume 4758 of *LNCS*, pages 179–194, 2007.
15. S. J. Owen, May 2007. http://www.andrew.cmu.edu/user/sowen/mesh.html.
16. J. Savolainen, I. Oliver, M. Mannion, and H. Zuo. Transitioning from Product Line Requirements to Product Line Architecture. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 186–195, 2005.
17. D. C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, Feb. 2006.
18. F. van der Linden. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4):41–49, 2002.