Reconciling Automation and Flexibility in Product Derivation

Gilles Perrouin, Jacques Klein, Nicolas Guelfi, Jean-Marc Jézéquel

Reconciling Automation and Flexibility in Product Derivation

Gilles Perrouin[†] Jacques Klein[*] Nicolas Guelfi[*] Jean-Marc Jézéquel[†]

[*]Laboratory for Advanced Software Systems,
University of Luxembourg,
6 rue Richard Coudenhove-Kalergi,
L-1359 Luxembourg-Kirchberg, Luxembourg
{jacques.klein,nicolas.guelfi}@uni.lu

[†]Triskell Team
IRISA/INRIA Rennes
Campus de Beaulieu
35042 Rennes, France
{gilles.perrouin,jezequel}@irisa.fr

Abstract

Product derivation, i.e. reusing core assets to build products, did not receive sufficient attention from the product-line community, yielding a frustrating situation. On the one hand, automated product derivation approaches are inflexible; they do not allow products meeting unforeseen, customer-specific, requirements. On the other hand, approaches that consider this issue do not provide adequate methodological guidelines nor automated support.

This paper proposes an integrated product derivation approach reconciling the two views to offer both flexibility and automation. First, we perform a pre-configuration of the product by selecting desired features in a generic feature model and automatically composing their related product-line core assets. Then, we adapt the pre-configured product to its customer-specific requirements via derivation primitives combined by product engineers and controlled by constraints that flexibly set product line boundaries. Our process is supported by the Kermeta metamodeling environment and illustrated through an example.

1. Introduction and motivation

Product Derivation (PD) [11] is the complete process of constructing a product from Software Product Line (SPL) core assets [1]. Since the final goal of SPLs is to enable organizations to deliver quality products within shortened development cycles, we could assume that PD is key to SPL approaches and has been exhaustively studied in order to unleash the full potential of the product-line paradigm. In fact, this was not the case and research [10] has shown that this process can be tedious and error-prone. Therefore, several automated product derivation approaches [44, 18, 23, 19, 8] have been proposed to assist product engineers in this task; most of them use model-driven techniques to derive products according to choices made by product engineers on the basis of a decision model. Such techniques help product engineers obtaining products reliably and decrease product development time thus minimizing costs. However, to be successful, a product derivation process should also be able to address SPL's customers specific and unanticipated requirements. These requirements may be only partially addressed by the SPL's core assets. As an example, we consider the `ItemCatalog` SPL [9], which will be used throughout this paper to illustrate our approach. It consists of a catalog for an e-commerce application as shown in Figure 1. A catalog has a structure (illustrated by the root feature `catalogStructure`) which contains descriptions of items (`ProductInformation`) and possibly an organization of these items in terms of `Categories`. Information concerning an item can be completed (via the optional feature `AssociatedAssets`) by media files. Media files considered are `2DImage` or `3DImage` and each item can be associated with only one of these media types. The optional `Categories` feature has four op-

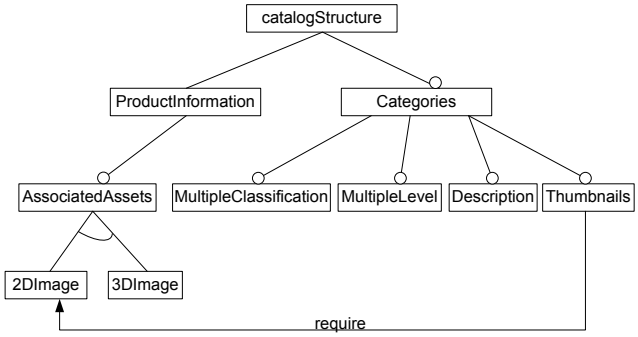

Figure 1. Product Catalog Feature Diagram

[1]According to Withey [42], an asset is: "a description of a partial solution (such as a component or design documents) or knowledge (such as a requirements database or test procedures)"

tional sub-features denoting the type of category being considered. `MultipleClassification` allows an item to pertain to several categories, `MultipleLevel` represents the support for nested categories, `Description` represents the ability for a category to own a description. Finally `Thumbnails` allows to sort items according to a reduced version of their images. In that case, the item has to be associated to `2DImage`, as depicted by the "requires" constraint.

Let us assume that a specific customer is interested in having such a catalog for its e-commerce application. In the following, this catalog will be referred to as `VideoCatalog`. `VideoCatalog` application supports simple categories which can contain the same item. Each category has a description. Additionally, `VideoCatalog` should offer video thumbnails rather than 2D pictures. To implement this catalog, the product engineer is faced to a dilemma. On the one hand, using automated product derivation for `ItemCatalog` will not allow to reach `VideoCatalog` because the "video" feature does not exist. On the other hand, most of the customer requirements coincide with `ItemCatalog` requirements and it would be time-consuming and error-prone to develop the product by reusing SPL's core assets manually. This situation is depicted in Figure 2: The innermost oval represents the set of products (illustrated by "V") explicitly identified by domain engineers and directly supported by automated derivation of SPL core assets. The outermost (dashed) oval represents products (noted by "O") which are both required to satisfy customers and sufficiently close to the SPL requirements so that reusing this SPL's core assets provide a significant advantage. Since customer-specific requirements cannot be anticipated, it is not possible to fully automate their support in the product. For example, `VideoCatalog` falls in this category of products. Finally, "X"-labeled products are too distant from the SPL. This distance might be measured in several units: functional/non-functional requirements adequacy, technical difference (the same requirements may have different implementations depending on the platform they are being supported) or marketing considerations. For example, `ProductInformation` may be used to store the catalog in a database. Building a catalog in which this feature has been significantly modified may imply many technical problems. Hence, such a product must be avoided.

To solve the product engineer's dilemma, we need to provide him with a PD approach which is *flexible* enough to allow him to derive "O"-labeled products *efficiently*. This means that he should be able to focus only on unforeseen customer requirements without having to worry about those which are directly supported via SPL's core assets derivation. Furthermore, we also need to prevent him from deriving "X"-labeled products which are out of the SPL's scope.

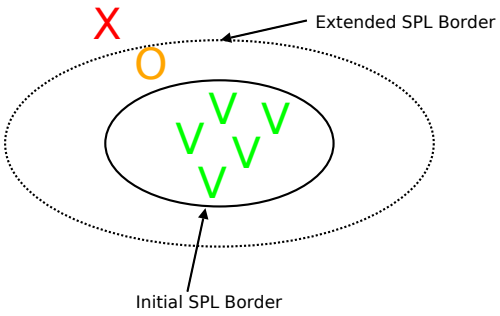

Figure 2. Flexible Product Derivation

Devising such a PD approach requires finding a trade-off between apparently conflicting efficiency and flexibility requirements.

This paper shows how, by combining an automated model composition approach with our earlier work on flexible product derivation [17, 32], a reconciliation is possible. First, we introduce a generic feature metamodel which supports a wide variety of existing feature models and which relates each feature to SPL's core assets UML models further defining and/or designing it. Based on product engineer's choices, we perform a pre-configuration of the product by automatically composing core assets thus resulting in a merged model. The second step of the derivation process consists in customizing this model to support customer-specific requirements. This customization is performed through a model transformation which is validated against OCL constraints defined on core assets. The whole approach is implemented on top of Kermeta, a general purpose metamodeling platform.

The remainder of the paper is structured as follows. Section 2 reviews existing PD approaches and sketches our vision on efficient and flexible PD. Section 3 describes our two-step PD process illustrated through our `ItemCatalog` example. Section 4 details our feature metamodel, UML elements used for core assets modeling as well as constraints ensuring feature diagram validation and controlling core assets customization. Section 5 explains how tool support is built in the Kermeta environment and details the derivation of `VideoCatalog`. Finally, Section 6 wraps up with conclusions and future work.

2. Background on model-driven product derivation

2.1. Model-driven engineering

Model-Driven Engineering (MDE) [22] advocates the use of *models* to face the inherent complexity of software systems. In [32], we considered a model as a set of statements defining an abstraction of a system (or the problem

addressed by that system) and fulfilling a particular purpose. Therefore, models help to reason and to communicate about software requirements and assets by simplifying and separating concerns. But MDE's ambition goes beyond mere software description. When precisely defined with respect to a metamodel (which provides an abstract syntax), models can be progressively refined all the way down to implementation by means of model transformations processing them. Models can be considered as primary building blocks for constructing software. Various MDE flavors have been developed to date, for example the OMG's Model Driven Architecture (MDA) [40] initiative which contributed to popularize MDE's ideas via standardization. We believe that MDE has a prominent role to play in product-line engineering to define their core assets and support product derivation. Current model-driven PD approaches can be organized into two main categories according to the derivation technique they use: *configuration* and *transformation*.

2.2. Product derivation by configuration

Product configuration or software mass customization [26, 27] originates from the idea that product derivation activities should be based on the parameterization and/or composition of the SPL core assets rather than focusing on how the individual products can be obtained. Most of these approaches base their decision models on *feature models* originally proposed by Kang et al. [20]. In an approach called FORM [21], Kang et al. define a derivation process starting with a requirements phase in which similar features to the features desired by a given customer are selected in the layers proposed (capabilities, operating environment, domain technologies and implementation techniques). Due to mapping between feature models and architecture as well as design artifacts, a selection on a feature model results in an already configured design model for the product. This refinement approach was later formalized by Czarnecki [7] through their concept of "staged configuration" [8]. Czarnecki and Antkiewicz [6] map feature models to UML activity and class diagrams via annotations. Annotations are associated with constraints mapped from the original feature model. This enabled them to define an automated configuration process for UML 2.0 models. In [41], Voelter and Groher propose to use model-driven and aspect-oriented development to support the derivation process; model transformations are used to provide a mapping between the problem domain (modeled as a feature model whose features are related to models further defining them) to the solution domain (defined using another modeling language). Aspect-oriented techniques support the actual core assets composition forming the products. Since its inception, product configuration has received an extensive commercial tool support [19, 35, 4]. Academic tools also exist [1].

2.3. Product derivation by transformation

An alternative approach to product derivation is to transform core assets rather than configuring them.

Haugen et al. [18] present a conceptual model for SPL engineering aligned with MDA standards. At the requirements elicitation level (corresponding to MDA's Computation Independent Model or CIM), the product line is modeled in terms of UML 2.0 use cases in a model called "product line model". A model transformation relates SPL's requirements to core assets (known as "system family model" corresponding to Platform Independent Model or PIM) modeled in terms of UML 2.0 composite structures. The system family model includes variability definition via stereotypes. Product Derivation proceeds as follows: first, the "product model" which is expressed using the same formalism as the product-line model is defined. Then, a model transformation taking both product and product line models as parameters transform the core assets so that the resulting model, "Product/System Model", correspond to the PIM model of the product. Using successive transformations, this model is finally implemented in the target platform. A similar approach has been proposed by Kim et al. [23].

A detailed transformational PD approach is given in [44]. Core assets are modeled in terms of UML 2.0 class diagrams for the static part and UML 2.0 sequence diagrams for the behavioral part. A profile to describe variability has also been defined [43]. OCL constraints have been defined in order to ensure consistency amongst variants. The decision model is a class diagram which exposes variants as stereotyped elements. Based on product engineer's choices, relevant classes are selected and a model transformation removes unused variants as well as optimizes the model. The behavioral part is derived by composition and formally defined using an algebraic approach.

2.4. Reconciling automation and flexibility

The aforementioned PD techniques address our efficiency requirement through automated support. However, we believe they fail on the flexibility requirement. Indeed, these techniques rely on the hypothesis that all customer requirements (belonging to the domain targeted by the SPL) can be identified by domain engineers and implemented via a direct combination of core assets ("V"-labeled products in Figure 2). Therefore they provide sophisticated ways to model explicitly variability in core assets to address customer requirements.

We believe this hypothesis to be false in the general case, yielding inappropriate approaches to cope with specific requirements such as the introduction of countless variants in the SPL's infrastructure (overly complexifying its management) or forcing it to evolve as a whole to address the

needs of one particular product. Some SPL methods have included a "product-specific" phase in their PD process, such as Kobra [2, 3] but do not provide any guidance for it. In [23], such a phase is called "integration": a transformation merges core assets models with specific model defined for the product, not much more is said about the nature of this transformation. In [41], Voelter and Groher propose to use aspects to weave product specific concerns into the configured model, however we cannot ensure that they are consistent with product line assets and scope.

Having identified this issue, we devised in previous work a flexible PD process [17], part of a model-driven SPL-based development methodology called FIDJI [32]. To guarantee flexible PD, FIDJI relies on the following principles to perform domain engineering. First, we propose to define core assets by restriction rather than exhaustively. Indeed, core assets models are defined with constraints (called *core assets instantiation constraints* and expressed in OCL) which prohibit illegal combination of core assets. For example, these constraints can enforce a mandatory dependency between two assets or prevent an asset from being changed while reused in a product. Therefore, these constraints act as a decision model characterizing the set of possible product line members without requiring to define them explicitly and enables unforeseen products to be developed provided they satisfy these constraints. Second, FIDJI separates the definition of core assets from instantiation constraints following an *orthogonal variability modeling* [33] approach. This results in simpler domain engineering models and a greater reuse potential for core assets (several SPLs can be defined with the same core assets but with different instantiation constraints).

The FIDJI PD process in itself consists in writing a model transformation, using a set of predefined transformation operations, that will reuse core assets' models to build the product. This transformation is written by the product engineer and checked against instantiation constraints. Hence, the FIDJI PD process offers the flexibility required to support product-specific requirements by supporting them via transformation operations while controlling their realization through instantiation constraints.

However, achieving flexibility was partially at the expense of automation. First, writing such a transformation can be complex. Second, FIDJI PD does not fully take advantage that in every "O"-labeled product (Figure 2), part of its elements comes from a "V"-labeled one. Complexity of writing the PD transformation can be reduced if we are able to automatically derive support for all the "V"-like features, leaving to the product engineer the responsibility to address "O"-like ones. To this end, we propose to improve the FIDJI PD process with a novel step, called *pre-configuration* which allows us to generate a skeleton of the product via core assets composition and based on a selection made on a feature model by the product engineer. This skeleton is then customized via a transformational approach so that it fits specific product requirements. This process is detailed in the next section.

3. Process

3.1. Pre-configuration

The pre-configuration step relies on the supply of two kinds of models (which will be fully detailed in Section 4) as a result of the domain engineering activity. First, we require a feature model which exposes in a concise way features and their variants supported by SPL's core assets. Second, core asset models (which are related to the features they support) have to be provided. These models need to be complemented with the following constraints:

- **Feature Modeling Constraints (FMC)**. These constraints are related to the correctness of the feature model itself (well-formedness rules). For instance, one of these rules states that the root feature has no parent.

- **Core Assets Instantiation Constraints (CAIC)**. They prohibit illegal combination of core asset models thus defining the extended SPL border (Figure 2). They should be compatible with choices offered by the feature model. For instance, CAIC controlling `ProductInformation`-related core asset models should not exclude composition with `AssociatedAssets` core asset models (e.g. preventing association between model elements to be defined),

The pre-configuration step is similar to configuration approaches presented in Section 1. The product engineer selects the features relevant for the product to be built. The set of selected features will be checked with respect to the feature model. If the selection is valid (with respect to the FMC), a skeleton of the product is generated by composing the core asset models related to the selected features.

At this point, CAIC can be checked on the product skeleton to highlight potential conflicts between core assets. This situation can arise if the feature model allows illegal combination of core assets and need to be reported to domain engineers so that they can solve the problem and provide the product engineer an updated feature model and/or core asset models. Indeed, at the domain engineering level, verifying the feature model with respect to CAIC implies to check each valid product. For any non-trivial feature diagram, combinatorial explosion would make intractable such a verification. The pre-configuration step ends when a viable product –i.e. issued from a valid combination of features and whose CAIC are not violated – is obtained.

Applied on VideoCatalog, the pre-configuration process will issue a product skeleton which is resulting from the combination of the following selected sub-features of catalogStructure: ProductInformation, AssociatedAssets, 2DImage for the "left branch" of the feature model depicted Figure 1 and Categories, Description and Thumbnails for the "right branch". The next paragraphs will explain how to customize this skeleton so that VideoCatalog requirements are met. See Section 5 for technical details.

3.2. Product customization

The customization process relies on the same transformation approach as the FIDJI PD process. It consists in writing a transformation that will complete and adapt the product skeleton to satisfy product specific requirements. Typically, it represents the step of transitioning from "V"-labeled products to "O"-labeled ones (Figure 2). Before writing the transformation, one needs to identify the features which will be concerned by the customization process. We use the configured feature model (i.e. the one showing decisions made by the product engineer during pre-configuration) to do so. In VideoCatalog, the only concerned feature is Thumbnails which need to be adapted to support video embedding. Moreover a new video feature has to be added to AssociatedAssets. The product engineer will have a look at the potential CAIC related to core assets supporting these features and write its transformation accordingly. The transformation approach is imperative, textual and based on Kermeta (see Section 5). A library of dedicated transformation operations is provided to ease transformation writing. Once the transformation is written, the customized product is checked against CAIC to validate its belonging to the SPL.

Over a fully manual adaptation of the SPL core assets, we believe such a transformation is worth the investment. First, the pre-configuration step greatly reduces the size of the transformation code to write. Second, traceability between core assets models and product models can be automatically computed, which helps determining the impact of an SPL evolution on a specific product. Finally, the transformation can be reused to support SPL evolution. Indeed, if the customer-specific requirements this transformation addresses are considered worthwhile to be included in the product line, these requirements can be directly supported by applying this transformation on the core assets. As opposed to SPL approaches such as ConIPF [19] that systematically adapt the core assets to support specific requirements, we believe this evolution should be performed on a per-case basis in order to control explicitly the core asset base.

4. Metamodels

In this section we describe the metamodels supported by our approach: a generic form of feature model used for the pre-configuration process and a subset of UML used for the definition of core assets transformed during the customization step.

4.1. Feature metamodel

As stated before, we chose to use feature models for their popularity and simplicity. They represent an ideal notation to represent the variability supported by SPL assets in a concise way. However, since their original definition by Kang et al.[20], a plethora of notations have been proposed ([7, 15, 21] to name a few). Indeed, feature models can be considered as a product line of notations sharing commonalities and exposing differences which are not always explicitly defined. In such a context, there is a risk of being dependent of a particular feature notation which is difficult to choose and unnecessarily restricts the applicability of our approach. Fortunately, Schobbens et al. [38, 39] performed a formal analysis of the existing feature notations. To do so, they developed a pivot abstract syntax called Free Feature Diagrams (FFDs) used to map any feature modeling construct found in existing notations in order to reason formally on the syntax and semantics of these notations. The universal nature of FFDs makes it suitable for various applications; we used it to reason on variability [16] and in the following, we will show how we can derive from FFDs a generic metamodel for feature model supporting any particular concrete syntax.

FFDs are defined in terms of a parametric structure whose parameters serve to characterize each FD notation variant. GT (Graph Type) is a boolean parameter indicating whether the considered notation is a Direct Acyclic Graph (DAG) or a tree. NT (Node Type) is the set of boolean operators available for this FD notation. These operators are of the form op_k with $k \in \mathbb{N}$ denoting the number of children nodes on which they apply to. Considered operators are and_k (mandatory nodes), xor_k (alternative nodes) or_k (true if any of its child nodes is selected), opt_k (optional nodes). Finally $vp(i..j)_k$ ($i \in \mathbb{N}$ and $j \in \mathbb{N} \cup *$) is true if at least i and at most j of its k nodes are selected. Existing other boolean operators can usually be expressed with vp. The union of $vp(i..j)_k$ is called $card$. GCT (Graphical Constraint Type) is the set of binary boolean functions that can be expressed graphically. A typical example is the "requires" between Thumbnails and 2DImage. Finally, TCL (Textual Constraint Language) tells if and how boolean constraints defined over the set of FD nodes can be defined. With the help of these sets, a generic abstract syntax for FDs is given. A FD is then composed of the following elements:

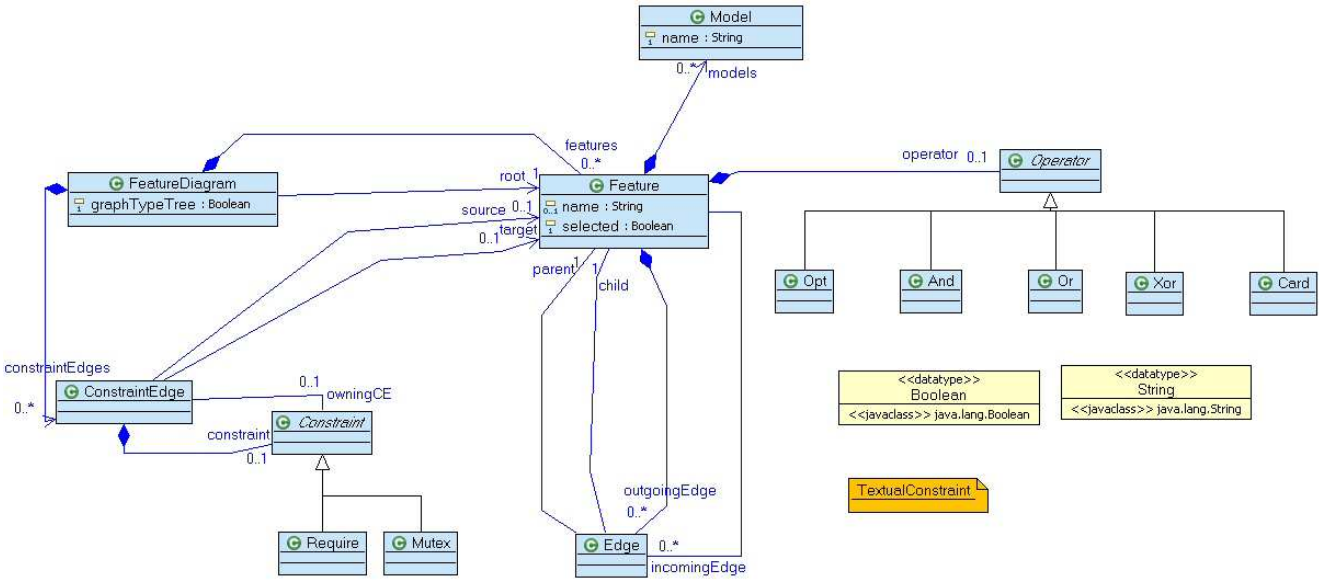

Figure 3. FFD-derived Metamodel

- A set of nodes N, which is further decomposed into a set of primitive nodes P (which have a direct interest for the product). Other nodes are used for decomposition purposes. A special root node, r represents the top of the decomposition (e.g. `catalogStructure` in our example),
- A function $\lambda : N \mapsto NT$ that labels each node with a boolean operator,
- A set $DE \in N \times N$ of decomposition edges. As FDs are directed, node $n1, n2 \in N, (n1, n2) \in DE$ will be noted $n1 \rightarrow n2$ where n1 is the *parent* and n2 the *child*,
- A set $CE \in N \times GCT \times N$ of constraint edges,
- A set $\phi \in TCL$

A FD has also some well-formedness rules to be valid: only root (r) has no parent; a FD is acyclic; if GT = true the graph is a tree; the arity of boolean operators must be respected.

These constructs were used to build our Ecore[2] based metamodel depicted in Figure 3. Its constitution was driven by simplicity and pragmatism. $FeatureDiagram$ is the root class of the metamodel. This class has an attribute $graphTypeTree$ corresponding to the boolean GT (Graph Type) presented previously. It also contains a list of features (class $Feature$) corresponding to the set of nodes N. The special root node r is identified by the reference $root$ from $FeatureDiagram$ to $Feature$. We decided to keep all the base operators (because they are simple and widely used) rather than using exclusively $card$ like operators. In the metamodel, these operators are subtype of the abstract class $Operator$, and each feature (class $Feature$) contains 0 or 1 operator (that corresponds to the function λ). The class $Feature$ also contains a list of edges (class $Edge$) allowing the construction of the set DE of decomposition edges. The set CE of constraint edges is represented in the metamodel by the class $ConstraintEdge$ and they are contained by the class $FeatureDiagram$. Each $ConstraintEdge$ contains either a $Require$ constraint or a $Mutex$ constraint. Primary feature nodes are related to UML models (see below) defining the core assets involved in the realization of these features. In the metamodel, a primary feature is related to UML models by the composite association between the class $Feature$ and the class $Model$. Finally, note that well-formedness rules (Feature Modeling Constraints) have been implemented in terms of constraints so that the conformance of FDs to our FFD-based metamodel can be checked in our tool (see Section 5).

[2]Ecore is a derivation of EMOF [31]

4.2. Asset metamodel

Each primary feature is related to a set of models defining core assets ensuring feature realization. For space reasons, in this paper, we will focus only on structural aspects though our product derivation approach is applicable to behavioral aspects as well [25, 24]. To model our assets we consider the subset of UML devoted to class diagrams. Methodologically, they can be used either at the requirements analysis level (also called *late requirements*) or at the design level. Figure 4 shows the class diagram modeling the asset associated to `Description`.

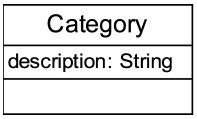

Figure 4. The optional "description" asset

CAIC are modeled using OCL 2.0 [30]. They take the form of invariants defined on the asset models. For example, the following invariant ensures that the name of a product as well as its price are kept:

```
Context productInformation::Product inv:
not self.name.oclIsUndefined()
and not self.price.oclIsUndefined()
```

5. Tool support

To validate the approach proposed in this paper, we have implemented it within the Kermeta environment[3]. This section is divided in four sub-sections. The first one presents the Kermeta environment and details our motivations for using it, while the second details Kompose, a model composition facility built on top of Kermeta. The two last subsections describe how the models associated to features are composed to form the product skeleton and how we can customize a product by means of kompose directives.

5.1. The Kermeta environment

Kermeta [28] is an open source meta-modeling language, designed as an EMOF extension. Kermeta extends EMOF with an action language that allows specifying metamodels' semantics. The action language is imperative and object-oriented. It includes both imperative Object Oriented (OO) features and model specific features. Kermeta includes traditional OO static typing, multiple inheritance and behavior redefinition/selection with a late binding semantics. To make Kermeta suitable for model processing, more specific concepts such as opposite properties (i.e. associations) and handling of object containment have been included. In addition to this, convenient constructions of the Object Constraint Language (OCL), such as closures (e.g. each, collect, select), are also available in Kermeta.

A complete description of the way the language was defined can be found in [28]. It was successfully used for the implementation of a class diagram composition technique in [36] but also as a model transformation language in [29]. To implement our feature model metamodel and the automatic product derivation proposed in this paper we have chosen to use Kermeta for several reasons.

First, Kermeta tools are compatible with the Eclipse Modeling Framework (EMF) [5] which allows us to use Eclipse tools to edit, store, and visualize models. In this way, our feature model metamodel being implemented with Kermeta, we can easily edit, store, load and visualize feature models conformed to our feature metamodel.

Second, FMC can be easily written and checked with Kermeta. Indeed, Kermeta allows the use of invariants on classes of a metamodel. An example of invariants for the metaclass `FeatureDiagram` in Figure 3 is: "a feature model is a graph or a tree which is acyclic". In the same way, these invariants can also be used to write and check CAIC. Constraints can be directly written in the Kermeta language or in OCL.

Third, the generic model processing abilities of the Kermeta language makes it suitable for our PD process which combines model composition and transformation. Our implementation use Kompose, an extension of Kermeta which provide high-level support for these tasks. Kompose is described in the next sub-section.

5.2. Kompose

Kompose [12, 13] is an Aspect-Oriented Modeling (AOM) approach based on the systematic merging of matching elements. Initially, AOM comes from the Aspect-oriented software development (AOSD) community. AOSD techniques aim to provide systematic means for the identification, separation, representation and composition of crosscutting concerns. Aspect-oriented ideas can be applied at any phase and at any level of abstraction during software development. AOM focuses on modularizing and composing crosscutting concerns at the model level.

Kompose generalizes the approach proposed by France et al. in [14, 37] in the context of class diagrams, for any metamodel. This approach supports merging of model elements that present different views of the same concept. The model elements to be merged must be of the same syntactic type, that is, they must be instances of the same metamodel class. An aspect view may also describe a concept that is not present in a target model, and vice versa. In these cases, the model elements are included in the composed model.

The process of identifying model elements to merge is called *element matching*. To support automated element matching, each element type (i.e., the element's meta-model class) is associated with a signature type that determines the uniqueness of elements in the type space: Two elements with equivalent signatures represent the same concept and thus are merged. A signature type is a set of syntactic properties associated with the element type. A model element's signature consists of the values associated with these prop-

[3]Our PD prototype as well as the full models defining our sample SPL can be downloaded at: `http://www.kermeta.org/mdk/ProductDerivation/`

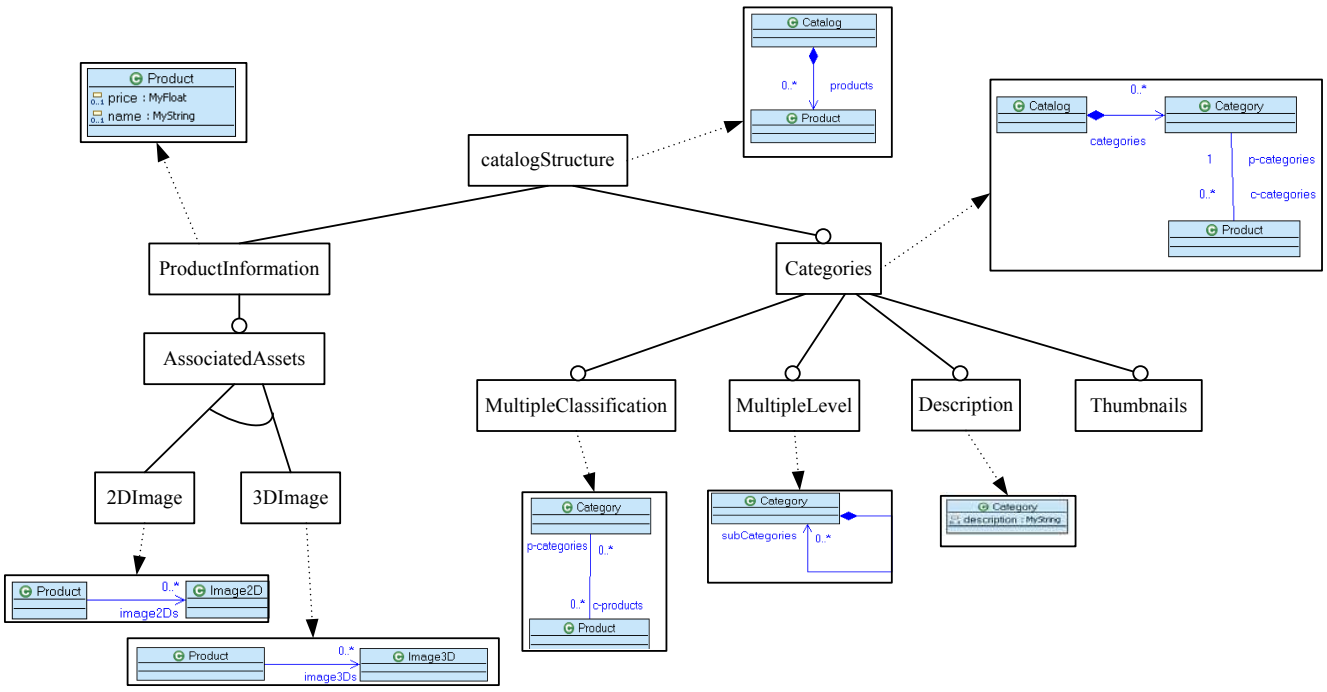

Figure 5. Example of core assets

erties. Currently, the signature of a model element consists only of its name; therefore, attributes and operations can be used to define different views of the same class. Attributes and operations match if and only if they have identical syntactic properties. Associations match if they have the same role names at their association ends.

Kompose allows users to specify *composition directives* that are used to ensure that the composed model also satisfies application specific properties [37]. Composition directives can be used to prepare models before merging (*pre-directives*) or after (*post-directives*). Examples of such directives include *Create* which creates new model elements, *Add* which adds them to other model elements, *Remove* and *Replace*.

In the rest of this section, we describe how Kompose has been used to support our two-step PD process.

5.3. Supporting pre-configuration with Kompose

As previously stated, in our approach a feature is related to a set of models defining core assets. Figure 5 presents the feature model of Figure 1 conforming to the FFD metamodel (Figure 3). Dashed arrows show the relationships between features and their realizing core asset models. The top of Figure 6 represents a subset of features corresponding to a valid product derived from the feature model in Figure 5. The bottom of Figure 6 presents the result of the merge performed with Kompose of the core assets related to the features of the derived product. This figure illustrates the fact that, for instance, classes with the same name are not duplicated but effectively merged. The product skeleton is automatically obtained from the feature model in Figure 5 by selecting the expected features and launching the derivation process written in Kermeta.

5.4. Product customization via Kompose directives

Once the product skeleton has been built by composing core assets thus completing the pre-configuration step, we need to customize this skeleton with respect to specific requirements. To do so we use kompose pre and post-directives which are applied on the composed product model. Kompose provides a simple textual language to combine directives. The support of `VideoCatalog` which requires to update `thumbnails` and to add a new `Video` class is expressed as follows:

```
Post {
 create Class as $c
 $c.name = "Video"
 catalogStructure.eClassifiers + $c
 catalogStructure::Product::
  thumbnail.eType = $c}
```

`Post` states that it is a post-directive block (i.e. processed after model composition). First line of the block uses a create directive to create class `$c` which is then named `Video`.

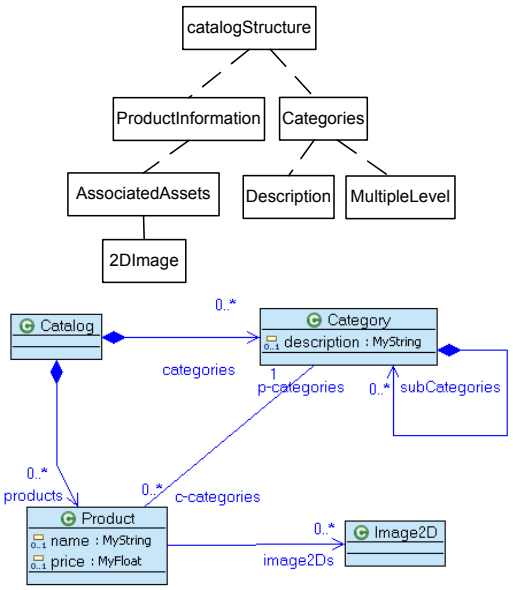

Figure 6. Result of the core assets composition for the derived product

This new class is added to the model as an element of `CatalogStruture` package containing the model of the product. Finally `thumbnail` is updated to support video. This file is transformed into a ".kompose" file instance of the Kompose directive metamodel [12]. This model is actually used by Kompose to process directives.

Our PD process ends with the verification of product validity with respect to the SPL. To do so we check relevant CAIC on the completed product model, this is the case here since we do not have made any change on product name and price attributes.

6. Conclusion

In this paper, we proposed a PD process which is a trade-off between automation and flexibility. We demonstrated how by combining well-known PD approaches, it is possible to provide tool support automating a significant part of this process. We also validated the utility of MDE in SPL engineering by showing how a generic metamodeling platform such as Kermeta is able to easily support the definition of domain-specific languages such as a generic feature metamodel, to provide conformance checks on models instances of this metamodel, and to provide any kind of behavior related to such DSLs (here model composition and transformation).

There is room for improvement. At the model level it may be interesting to study how our PD approach behaves with respect to SPL evolution. By using the same metamodels for domain and product engineering, the PD transformation could be reused without changes on the core assets to make the SPL evolve. In our context, in addition of the above we may have to complete the resolved feature model of the product and to merge it with the SPL feature model so that variability definition is kept consistent with core assets. At the tool level, improvements may concern the visual representation of feature models (now either editable via textual XML representation or via the Ecore reflexive editor provided by Eclipse). A possibility is to use tools such as TopCased [34] or GMF[4] to generate visual editors based on our metamodel.

Acknowledgements

The authors would like to thank Dr. Franck Fleurey for his valuable help on Kompose. This work has been partially supported by the S-Cube FP7 Network of Excellence.

References

[1] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] C. Atkinson, J. Bayer, and D. Muthig. Component-based Product Line Development: the KobrA approach. In *SPLC*, pages 289–309, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[4] BigLever. GEARS Website `http://www.biglever.com/index.html`, 2006.

[5] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, 2003.

[6] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach based on Superimposed Variants. In *Generative programming and component engineering (GPCE)*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[9] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, pages 211–220, New York, NY, USA, 2006. ACM.

[4] `http://www.eclipse.org/gmf/`

[10] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in Software Product Families: Problems and Issues during Product Derivation. In *SPLC*, pages 165–182, September 2004.

[11] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74(2):173–194, 2005.

[12] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In *Workshop on Aspect-Oriented Modeling at Models'07*, 2007.

[13] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *EDOC*, Annapolis, MD, USA, 2007.

[14] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings Software*, pages 173–185, August 2004.

[15] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *ICSR*, Washington, DC, USA, 1998.

[16] N. Guelfi and G. Perrouin. Coherent Integration of Variability Mechanisms at the Requirements Elicitation and Analysis Levels. In D. Muthig and P. Clements, editors, *Workshop on Managing Variability for SPL*, Baltimore, MD, USA, 2006.

[17] N. Guelfi and G. Perrouin. A flexible requirements analysis approach for software product lines. In *REFSQ*, LNCS-4542, pages 78–92, Norway, 2007. Springer-Verlag.

[18] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, and A. Solberg. An MDA-based Framework for Model-Driven Product Derivation. In *SEA*, pages 709–714. ACTA Press, 2004.

[19] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families, The ConIPF Methodology*. IOS Press, 2006.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Nov. 1990.

[21] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[22] S. Kent. Model Driven Engineering. In *IFM*, pages 286–298, London, UK, 2002. Springer-Verlag.

[23] S. D. Kim, H. G. Min, J. S. Her, and S. H. Chang. DREAM: A Practical Product Line Engineering Using Model Driven Architecture. In *Information Technology and Applications (ICITA)*, pages 70–75, Washington, DC, USA, 2005.

[24] J. Klein, F. Fleurey, and J. M. Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development (TAOSD).*, LNCS 4620:167–199, 2007.

[25] J. Klein, L. Hélouet, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Aspect-Oriented Software Development (AOSD)*, Bonn, Germany, 2006. ACM.

[26] C. W. Krueger. Easing the Transition to Software Mass Customization. In *Workshop on Software Product-Family Engineering (PFE)*, pages 282–293, London, UK, 2002. Springer-Verlag.

[27] C. W. Krueger. New Methods in Software Product Line Development. In *SPLC*, pages 95–102. IEEE, 2006.

[28] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML'2005*, LNCS, Jamaica, 2005. Springer.

[29] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop (MTIP)*, Jamaica, 2005.

[30] OMG. UML 2.0 OCL 2.0 specification. Technical Report ptc/05-06-06, Object Management Group, June 2005.

[31] OMG. Meta object facility (mof) core specification. Technical Report 06-01-01, OMG, January 2006.

[32] G. Perrouin. *Architecting Software Systems using Model Transformation and Architectural Frameworks*. PhD thesis, FSTC, Université du Luxembourg, and Institut d'Informatique, Université de Namur, Sept 2007.

[33] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[34] N. Pontisso and D. Chemouil. Topcased combining formal methods with model-driven engineering. In *International Conference on Automated Software Engineering (ASE)*, pages 359–360, Washington, DC, USA, 2006.

[35] PureSystems. Pure::Variants Website `http://www.pure-systems.com/`, 2006.

[36] R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry. Model composition - a signature-based approach. In *AOM Workshop*, Montego Bay, Oct. 2005.

[37] R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development (TAOSD)*, LNCS 3880:75–105, 2006.

[38] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE*, Minneapolis, Minnesota, USA, 2006.

[39] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[40] R. Soley and OMG. Model Driven Architecture. Technical Report omg/00-11-05, OMG, November 2000.

[41] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC*, pages 233–242, Washington, DC, USA, 2007.

[42] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, ADA 315653, Software Engineering Institute, 1996.

[43] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML Profile for Software Product Lines. In *Product-Family Engineering (PFE)*, volume 3014 of *LNCS*, pages 129–139, Siena, Italy, November 2003. Springer.

[44] T. Ziadi and J.-M. Jézéquel. Product Line Engineering with the UML: Deriving Products. In *Families Research Book*. Springer, 2006.