Automatic Tool Support for Cardinality-based Feature Modeling with Model Constraints for Information Systems Development

Abel Gómez and Isidro Ramos

Abstract Feature Modeling is a technique that uses diagrams to characterize the variability of software product lines. The arrival of metamodeling frameworks in the Model-Driven Engineering field (MDE) has provided the necessary background to exploit these diagrams (called feature models) in information systems development processes. However, these frameworks have some limitations when they must deal with software artifacts at several abstraction layers. This paper presents a prototype that allows the developers to define cardinality-based feature models with complex model constraints. The prototype uses model transformations to build Domain Variability Models (DVM) that can be instantiated. This proposal permits us to take advantage of existing tools to validate model instances and finally to automatically generate code. Moreover, DVMs can play a key role in complex MDE processes automating the use of feature models in software product lines.

1 Introduction

The changing nature of technology and user requirements leads us to need multiple versions of the same or similar software application in short time periods. The Software Product Line (SPL) [5] concept arises with the aim of controlling and minimizing the high costs of developing a family of software products in the previous context. This approach is based on the creation of a design that can be shared among all the members of a family of programs within an application domain. The key aspect of Software Product Lines (SPL) that characterizes this approach with respect

Abel Gómez
Department of Information Systems and Computation, Universidad Politécnica de Valencia, Cno. de Vera s/n. Valencia, Spain. e-mail: `agomez@dsic.upv.es`

Isidro Ramos
Department of Information Systems and Computation, Universidad Politécnica de Valencia, Cno. de Vera s/n. Valencia, Spain. e-mail: `iramos@dsic.upv.es`

to other software reuse techniques is how to describe and manage variability, mainly by means of feature modeling as proposed in [12].

The use of feature models can be exploited by means of metamodeling standards. In this context, the Model-Driven Architecture [14] proposed by the Object Management Group is a widely used standard which arises as a suitable framework for this purpose. In this environment, MDE and the Generative Programming approach [6] provides a suitable basis to support the development of SPLs. Moreover, Generative Programming and SPLs facilitate the development of software products for different platforms and technologies.

In this paper we present a prototype that uses nowadays metamodeling tools to define cardinality-based feature models and their configurations. The selected platform is the Eclipse Modeling Framework (EMF) [9]. Moreover, feature models can be enriched with complex model constraints that can be automatically checked by means of the internal OCL interpreter. All these features of EMF allows developers to start a Software Product Line.

The remainder of this paper is structured as follows: in section 2 we briefly introduce feature modeling and in 3 we describe how nowadays standards and tools can be used to exploit them. In section 4 our proposal is presented. Related works are discussed in section 5 and in section 6 we present our conclusions.

2 Feature models and cardinality-based feature models at a glance

Feature models are diagrams which express the commonalities and varibilities among the products of a SPL. These models organize the so-called features (*user-visible aspect or characteristic of the domain*) in a hierarchical structure. The basic relationships between a feature and its children are: *mandatory* relationships (which represent de shared design), *optional* relationships, *OR* groups and *XOR* groups.

Cardinality-based feature modeling [7] integrates several of the different extensions that have been proposed to the original FODA notation [12]. A cardinality based feature model is also a hierarchy of features, but the main difference with the original FODA proposal is that each feature has associated a *feature cardinality* which specifies how many clones of the feature are allowed in a specific configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured. Moreover, features can be organized in *feature groups*, which also have a *group cardinality*. This cardinality restricts the minimun and the maximun number of group members that can be selected. Finally, an *attribute type* can be specified for a given feature. Thus, a primitive value for this feature can be defined during configuration which is useful to define *parameterized features*.

In feature models is also quite common to describe constraints between features such as the *implies* and the *excludes* relationships, which are the most common used ones. In classic feature models the semantics of these constraints can be expressed

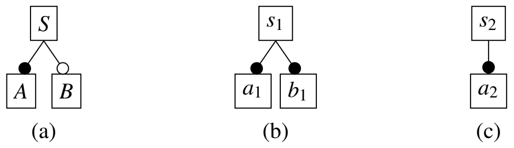


Fig. 1 Example of a feature model (1a) and the two possible configurations that it represents (1b and 1c).

by means of propositional formulas [2], thus, it is possible to reason about the satisfiability of the feature model and its configurations.

A configuration of a feature model can be defined as *a valid set of instances of a feature model*. I.e., the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. In Fig. 1a an example feature model is represented. This feature model represents a system S, with two features A and B. The first one, feature A, is mandatory (it must be included in every possible product of the product line), and the second one, feature B, is optional (it can be included in a particular product or not). Thus, we have two possible configurations for this feature model, which are represented in figures 1b and 1c.

3 Feature Modeling and Model–Driven Engineering

Model-Driven Engineering (MDE) is a Software Engineering field that over the years has represented software artifacts as models in order to increase productivity, quality and to reduce costs in the software development process. Nowadays, there is increasing interest in this field, as demonstrated by the OMG guidelines that support this trend with the Model-Driven Architecture (MDA [14]) approach.The Meta Object Facility standard (MOF, [16]), which provides support for meta-modeling,

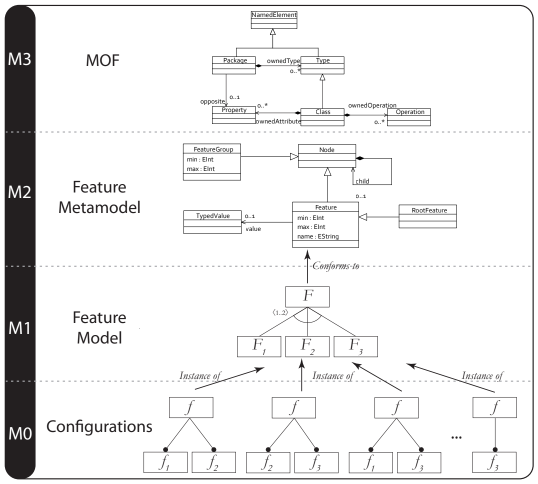


Fig. 2 Definition and configuration of feature models in the context of MOF. The EMOF language is represented in a simplified way in the level M3. In the level M2 the metamodel for cardinality-based feature models is represented by using the MOF language (also in a simplified way). In the level M1 feature models are described. Some configurations of the example feature model are shown at level M0.

defines a strict classification of software artifacts in a four-layer architecture (from M3 to M0 layer). As it provides support for modeling and metamodeling, we can use MOF to define cardinality-based feature models by defining its metamodel. Thus, we can define a model which captures the whole variability of the domain, which, in turn, allows us to define any possible configuration of the model. But, what is more, it can also be used to define model-based transformations that enable the use of feature models and their configurations in other complex processes. Fig. 2 shows where feature models and configurations fit in the four-layer MOF architecture.

3.1 MDA in practice: industrial tool support

The Eclipse Modeling Framework (EMF) [9] can be considered as an implementation of the MOF architecture. Ecore, its metamodeling language, can be placed at layer M3 in the four-layer architecture of the MOF standard. By means of Ecore, developers can define their own models which will be placed at the metamodel layer (M2). An example of such metamodels is the one to build cardinality-based feature models. Finally, these Ecore models can be used to automatically generate graphical editors which are capable of building *instance models*, which will be placed at M1 layer. In the case of feature modeling, these *instance models* are the feature models. The left column on Fig. 3 shows this architecture.

As can also be seen in Fig. 3, the M0 layer is empty. This is a limitation of most of the modeling frameworks which are available today. As said, EMF provides a modeling language to define new models and their instances, but this framework only covers two layers of the MOF architecture: the metamodel and the model layers. However, in the case of feature modeling we need to work with three layers of the MOF architecture: metamodel (cardinality-based feature metamodel), model (cardinality-based feature models), and instances (configurations).

Fig. 3 shows how to overcome this drawback: it is possible to define a model-to-model transformation in order to convert a feature model (i.e. the model represented by Feature model which can not be instantiated) to an Ecore model (i.e. the Domain Variability Model, DVM, which represents the Feature model as a new class dia-

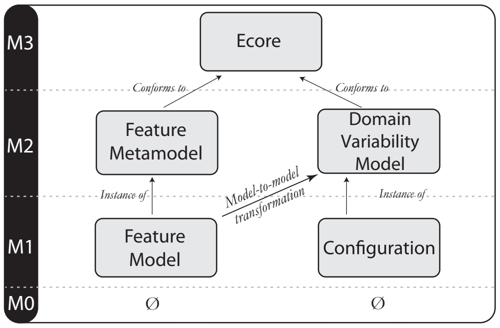


Fig. 3 EMF and the four-layer architecture of MOF. The Ecore language is placed at the M3 layer, the Ecore models (such as the metamodel for cardinality-based features models) are placed at the M2 layer, and model instances (feature models) are placed at the M1 layer. It is noteworthy that EMF can not represent more than 3 layers.

gram). Thus, it is possible to represent a feature model at the metamodeling layer, making the definition of its instances possible. This way, developers can take advantage of EMF again, and automatically generate editors to define feature model configurations, and validate them against their corresponding feature models thanks to their new representation, the DVM. Moreover, as the DVM is an Ecore model (a simplified UML class diagram) we automatically obtain support to check complex constraints (by using OCL) over the feature model configurations.

4 Our approach

Based on the concepts presented in the previous section and using EMF, we have developed a tool that allows us to automate several steps in order to prepare a feature model that can be exploited to develop a SPL in the context of MDA. In this sense, our tool provides:

- Graphical support to define (a variant of) cardinality-based feature models with model constraints expressed by using a constraint language.
- Support to automatically generate DVMs from feature models that capture all the variability of the application domain (including complex model constraints), allowing the developers to use them in model transformations.
- Support to transform model constraints to OCL expressions.
- Configuration editors, which will assist the developers.
- Capabilities to check the consistency of a configuration against its corresponding feature model by using pre-built OCL engines.

All these tasks are automatically supported by using MDE techniques (modeling, metamodeling, model transformations and code generation). The following subsections describe how this process has been implemented.

4.1 Cardinality-based feature metamodel

The basis of our work is the cardinality-based feature metamodel, which permits to define feature models. Fig. 4 shows our feature metamodel. Such metamodel has been defined taking into account that every element will have a different graphical representation. This way, it is possible to automatically generate the graphical editor to draw feature models based on such metamodel.

4.1.1 Feature models structure

In Fig. 4, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features, the set of Relationships among

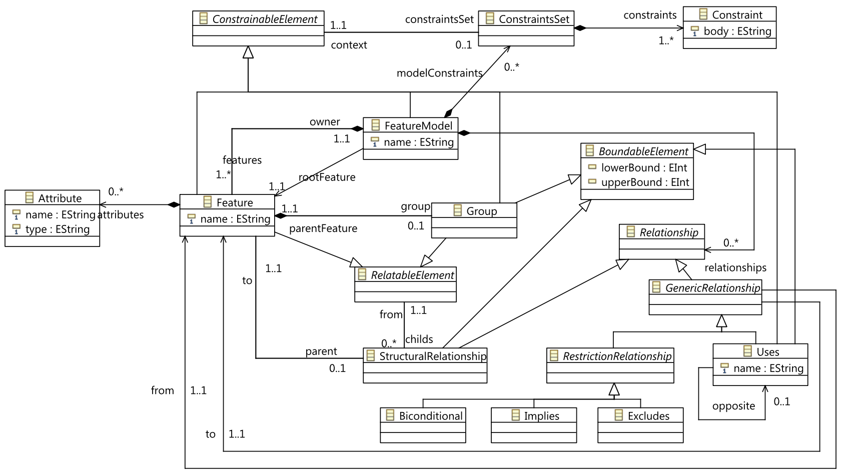


Fig. 4 Cardinality-based features metamodel

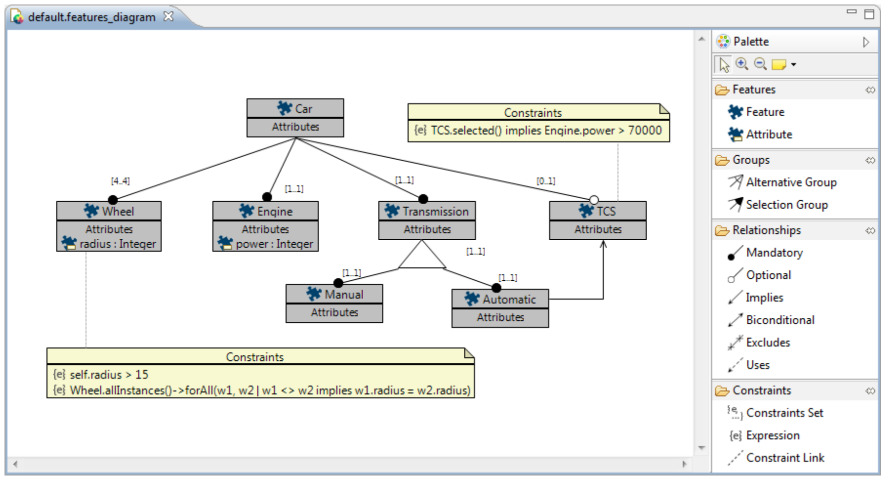
them and the set of model constraints (modelConstraints role) that are applied to it. A feature model must also have a root feature, which is denoted by means of the rootFeature role. As can be seen in Fig 4, our proposal represents explicitly the relationships between features. Thus, it represents in an uniform way the hierarchical relationships (StructuralRelationship class) and the restrictions between features (RestrictionRelationship class). The classification of these relationships is explained in detail in [10].

4.1.2 Feature model constraints

As was pointed out in section 2, it is quite common in feature modeling to have the possibility to define model constraints in order to describe more precisely which configurations should be considered as valid. Typically, these constraints are described by means of implication or exclusion relationships. This kind of relationships are the *binary and horizontal relationships* that our metamodel provides.

The *binary and horizontal relationships* are specified between two features and they can express constraints (coimplications, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features, however, the second one allows us to define dependencies at instance level, i.e.:

- *Implication ($A \longrightarrow B$)*: If an instance of feature A exists, at least an instance of feature B must exist too.
- *Coimplication ($A \longleftrightarrow B$)*: If an instance of feature A exists, at least an instance of feature B must exist too and vice versa.
- *Exclusion ($A \times\!\!\longrightarrow\!\!\times\ B$)*: If an instance of feature A exists, can not exist any instance of feature B and vice versa.



```
{e} TCS.selected() implies Engine.power > 70000
```

```
{e} self.radius > 15
{e} Wheel.allInstances()->forAll(w1, w2 | w1 <> w2 implies w1.radius = w2.radius)
```

Fig. 5 Example feature model

- *Use* ($A - - \rightarrow B$): This relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).

Besides these kind of relationships that describe coarse-grained restrictions, our metamodel provides capabilities to describe fine-grained restrictions. These restrictions are stored in a Constraints Set instance, and can be applied to any subclass of the abstract class ConstrainableElement (context role), i.e., FeatureModel, Feature, Group or Uses. The restrictions are expressed as a textual expression (body attribute of the Constraint class).

To describe these fine-grained restrictions we propose a constraint language, called *Feature Modeling Constraint Language* (FMCL). FMCL is a formal language without side-effects (does not modify the model instances) whose syntax is based on the widely known *Object Constraint Language* (OCL) and its semantics are defined by a set of patterns that describe the equivalences between FMCL expressions and OCL expressions.

4.1.3 Cardinality-based feature modeling editor

Following the Model-Driven Software Development (MDSD) approach, graphical editors can be automatically generated from the metamodel presented: i.e., the cardinality-based feature modeling editor. This editor allows us to easily define new feature models. To obtain this graphical editor, the Graphical Modeling Framework (GMF [8]) has been used.

Fig. 5 shows what this editor looks like. The palette is located on the right side of the figure, and shows the tools that can be used to define the feature models. In the

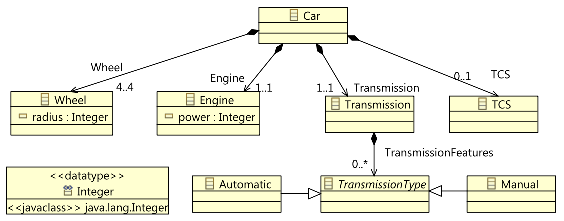


Fig. 6 Generated DVM for the example feature model. In the figure, model constraints (OCL annotations) have been omitted for clarity purpouses.

canvas an example feature model is shown. This feature model describes a simple product line for cars. A car must have four wheels (of a given radius), one engine (of an specific power in watts) and a transmission (which can be manual or automatic). As an optional equipment the car can have a Traction Control System (TCS). The feature model also describes four constraints: the arrow between the feature TCS and Automatic states that if an automatic transmission is selected, the TCS must be selected too; the annotation attached to the TCS feature states that the TCS can only be selected if the power of the engine is higher than 70.000 watts; and finally, the annotation attached to the Wheel feature specifies that the radius of the instances of the wheel must be higher than 15 inches and that all the wheels must be of the same size.

4.2 The Domain Variability Model

The Domain Varibility Model is a class diagram (an Ecore model) whose instances are equivalent to the configurations of a feature model. It is intended to ease the definition of feature models configurations in EMF as was explained in section 3.1. This model can be automatically generated by means of a model-to-model transformation. Following the MDA guidelines, this transformation is defined by using the Relations language defined in the QVT standard [15]. In order to integrate and execute this transformation process in our prototype, a custom tool based on the mediniQVT [11] transformations engine has been built.

4.2.1 The structure of the DVM

As can be observed in Figs. 5 and 6, the transformation regarding to the structure of the DVM is almost a one-to-one mapping. For each Feature of the source model an *EClass* (with the same name) is created. All the classes are created inside the same EPackage, whose name and identifier derives from the feature model name. Moreover, for each feature Attribute, an EAttribute in its corresponding EClass is created in the target model. Any needed EDataType is also created.

Regarding to the relationships, for each StructuralRelationship from a parent Feature, a *containment* EReference will be created from the corresponding EClass and for each Group contained in a Feature a *containment* EReference will be created from the corresponding EClass. This EReference will point to a new abstract class, whose name will be composed by the Feature name and the suffix “Type”. Additionally, an EClass will be generated for each Feature belonging to a Group. Moreover, each one of these EClasses inherit from the abstract *EClass* that has been previously created. Finally, for each Uses relationship between two Features, an EReference will be created in the target model. This EReference will relate two EClasses whose names will match the Features names.

4.2.2 Constraints over the DVM

The *restriction relationships* and *model constraints* (FMCL expressions) are mapped to OCL expressions in the DVM. The mappings to transform the *restriction relationships* to OCL expressions is described in [10] in detail.

The FMCL expressions are mapped to OCL expressions taking into account the mappings explained in section 4.2.1. Fig. 5 shows an example of this. As can be seen on the constraint that applies to the Wheel feature, a FMCL expression can be expressed directly using the OCL syntax. This way, an FMCL expression is directly transformed to an OCL invariant. The context of the invariant corresponds to the name of the *ConstrainableElement* that is linked to the constraint (dashed line in the figure) and the text of the expression remains the same:

```
context Wheel
inv: self.radius > 15
inv: Wheel.allInstances()->forAll(w1, w2 |
                              w1 <> w2 implies w1.radius = w2.radius)
```

However, although the FMCL expressions are almost the same than an OCL invariant, some simple conventions have been adopted to make the definition of model constraints closer to the feature modeling context. The semantics of these additions are defined by means of transformation patterns (see Table 1).

An example of the application of some of these patterns can be seen in the constraint attached to the TCS feature (Fig. 5). The example constraint is transformed to the following OCL expression:

```
context TCS inv:
TCS.allInstances()->notEmpty() implies Engine.allInstances()->forAll(power >
     70000)
```

Table 1 Summary of transformation patterns (FMCL to OCL)

FMCL expression pattern	Equivalent OCL definition
ConstrainableElement	*ConstrainableElement*`.allInstances()`
ConstrainableElement`.`*property op expression*	*ConstrainableElement*`.allInstances()` `->forAll(`*property op expression*`)`
ConstrainableElement`.selected()`	*ConstrainableElement*`.allInstances()` `->notEmpty()`
FeatureName`.childs()`	*FeatureName*`Type.allInstances()`

4.3 Creating and validating configurations

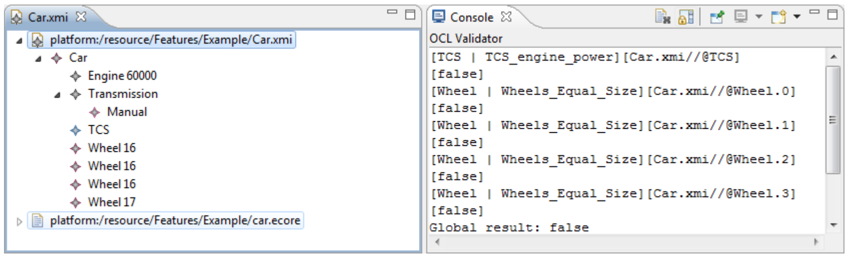
In order to create new configurations of feature models it is not necessary to use any custom tool. As far as we have a DVM which captures the same variability than the original feature model, developers can use the standard Ecore tools. However, in EMF there is not a default tool to check OCL invariants which are directly stored as EAnnotations in Ecore models themselves. Thus, we have built an extension which can take advantage of the OCL invariants that have been automatically created in the previous transformation step.

Fig. 7 shows an example configuration. It shows a car configuration with manual transmission, TCS, 4 wheels and engine. The radius of three of the wheels is 16 inches, and the radius of the fourth is 17 inches. The power of the engine is 60.000 watts. This configuration is invalid conforming to the restrictions applied to the metamodel. When the configuration is invalid the checking process is unsuccessful. In this situation, the prototype console shows a summary with the constraints that are not met, and which are the problematic elements as the figure shows.

5 Related works

Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals for variability management have arisen. Most of them are based in the original FODA notation and propose several extensions to it [4]. Our work is closely related with previous research, however, there are several distinctive aspects:

Our work describes a prototype to define and validate configurations of feature models. Previous work has been also done in this area, such as the *Feature Modeling Plugin* [1]. The main difference with our work is that configurations are defined in terms of the feature metamodel and both models and configurations coexist at the same layer. Thus, in order to be able to deal both with models and configurations it is necessary to build complex editors (as they must guarantee that the specialization



```
platform:/resource/Features/Example/Car.xmi
  Car
    Engine 60000
    Transmission
      Manual
    TCS
    Wheel 16
    Wheel 16
    Wheel 16
    Wheel 17
platform:/resource/Features/Example/car.ecore
```

```
OCL Validator
[TCS | TCS_engine_power][Car.xmi//@TCS]
[false]
[Wheel | Wheels_Equal_Size][Car.xmi//@Wheel.0]
[false]
[Wheel | Wheels_Equal_Size][Car.xmi//@Wheel.1]
[false]
[Wheel | Wheels_Equal_Size][Car.xmi//@Wheel.2]
[false]
[Wheel | Wheels_Equal_Size][Car.xmi//@Wheel.3]
[false]
Global result: false
```

Fig. 7 Example of an unsuccessful configuration check.

process is properly done), and whats more, those artifacts can not be easily used in complex MDE processes.

Some previous works have already represented feature models as class diagrams ([7] and [13]). However, in [7] no set of transformation rules defines the mappings between features and classes. In this work, OCL is also presented as a suitable approach to define model constraints, but there is no automatic generation of OCL invariants as the transformation is not clearly defined. In turn, [13] presents a set of QVT rules to define the mappings. However, in this case, neither model constraints nor configuration definitions support is presented.

In [2] a proposal for feature constraints definition and checking is done, representing features as propositions and restrictions among them as propositional formulas. However, this approach is not suitable when feature can have typed attributes which can not be expressed by this kind of formulas. We state that more expressive languages are needed, such as FMCL/OCL.

6 Conclusions

In this paper we have presented a framework[1] to define and use feature models in a MDE process. This framework addresses two issues: first, the incapability of nowadays metamodeling tools to deal simultaneously with artifacts located in all the MOF layers; and second, the complexity to define model constraints in feature models where features can be cloned and can have attributes. These problems have been solved by transforming feature models to Domain Variability Models that can be instantiated and reused in future steps of the MDE process.

Our tool has been designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. By means of generative programming techniques, a graphical editor for feature models has been built. Feature models defined with this editor are automatically transformed in DVMs that are used to define configurations of feature models. Although several tools to define feature models and configurations in the last years have arised, our approach has several advantages against previous approaches: (i) the infrastructure that we propose to build configurations is simpler and more maintainable, as it is built following the MDSD guides; (ii) configurations are actually instances of a feature model (expressed by means of the DVM), so we can take advantage of the standard EMF tools; (iii) as feature models are described by DVMs that can be instantiated, both models and configurations can be used in other MDE tasks; (iv) having a clear separation between feature models and configuration eases the validation tasks as they can be performed by means of built-in languages; and (v) as the transformation between feature models and DVMs is performed automatically by means of a declarative language we can trace errors back from DVMs to feature models.

[1] This framework is supported by a prototype that can be downloaded from `http://issi.dsic.upv.es/~agomez/feature-modeling`.

It is noteworthy to remark the importance of using feature models and configurations at different layers. In [3] an example where this architecture is used to integrate feature models in a MDE process is shown. This work describes how a model transformation with multiple inputs (feature models and functional models) is used to generate a software architecture automatically.

Acknowledgements This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

References

1. M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, 2004.
2. D. Batory. Feature models, grammars, and propositional formulas. pages 7–20. Springer, 2005.
3. M. E. Cabello, I. Ramos, A. Gómez, and R. Limón. Baseline-oriented modeling: An mda approach based on software product lines for the expert systems development. *Intelligent Information and Database Systems, Asian Conference on*, 0:208–213, 2009.
4. L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, 2009.
5. P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.
6. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
7. K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.
8. Eclipse Organization. The Graphical Modeling Framework, 2006. `http://www.eclipse.org/gmf/`.
9. EMF. `http://download.eclipse.org/tools/emf/scripts/home.php`.
10. A. Gómez and I. Ramos. Cardinality-based feature modeling and model-driven engineering: Fitting them together. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'10)*, Linz, Austria, Jan. 2010.
11. ikv++ technologies AG. ikv++ mediniQVT website. `http://projects.ikv.de/qvt`.
12. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.
13. M. A. Laguna, B. González-Baixauli, and J. M. Marqués Corral. Feature patterns and multi-paradigm variability models. Technical Report 2008/01, Grupo GIRO, Departamento de Informática, may 2008.
14. Object Management Group. MDA Guide Version 1.0.1. 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.
15. Object Management Group. *MOF QVT Final Adopted Specification*, June 2005.
16. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.