Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together

Abel Gómez[1], Isidro Ramos[2]
Department of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
[1]agomez@dsic.upv.es
[2]iramos@dsic.upv.es

***Abstract*—Feature Modeling is a technique which uses a specific visual notation to characterize the variability of product lines by means of diagrams. In this sense, the arrival of metamodeling frameworks in the Model-Driven Engineering field has provided the necessary background to exploit these diagrams (called feature models) in complex software development processes. However, these frameworks (such as the Eclipse Modeling Framework) 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 constraints. These models are automatically translated to Domain Variability Models (DVM) by means of model-to-model transformations. Thus, such models can be instantiated, and each different instantiation is a configuration of the feature model. This appproach allows us to take advantage of existing generative programming tools, query languages and validation formalisms; and, what is more, DVMs can play a key role in MDE processes as they can be used as inputs in complex model transformations.**

***Keywords*-Software Product Lines; Model Driven Architecture; Feature Modeling; UML; OCL**

I. Introduction

The key aspect of Software Product Lines (SPL) [6] that characterizes this approach against other software reuse techniques is how to describe and manage variability. Although several approaches have addressed this problem, the most of them are based on feature modeling, proposed in [12]. In this approach, the commonalities and varibilities among the products of a SPL are expressed by means of the so-called features (*user-visible aspect or characteristic of the domain*), which are hierarchically organized in feature models.

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. In this sense, the Meta Object Facility (MOF) and the Query/Views/Transformations (QVT) standards allows us to define feature models and their instances, and use them in a Model-Driven Engineering (MDE) process. In this context, MDE and the Generative Programming approach [7] 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 their use under different technologies.

In this paper we discuss the main issues that arise when trying to use feature models in a MDE process, and how to easily overcome them. We also present a tool that allows the developers of SPLs to define, use and exploit feature models in a modeling and metamodeling tool. In our case, we have chosen the Eclipse Modeling Framework (EMF). Moreover, the EMF framework provides several tools which permit us to enrich these models (by means of OCL expressions and OCL interpreters) and to deal easily with them (by using model transformations). All these features of EMF allows us to use this framework to start a Software Product Line.

The remainder of this paper is structured as follows: in section II we present the starting point of our work and the main problems that arise when trying to use feature models in a MDA process; and in sections III and IV we present both the ideal approach and the practical approaches to use feature modeling in a MDA context. In section V we show our feature metamodel and how we operationalize the solution presented in section IV in our prototype tool. Related works are discussed in section VI and in section VII we present our conclusions and future works.

II. Foundations

A. Cardinality-based feature models at a glance

Cardinality-based feature modeling [8] integrates several of the different extensions that have been proposed to the original FODA notation [12]. In this sense, 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.

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. According to the original FODA notation, these constraints can be expressed by means of propositional formulas [2], thus, it is possible to reason about the satisfiability of the feature model. As explained in [8], this interpretation for feature models is not very adequated when dealing with cardinality-based feature models due to the fact that we can have multiple copies of the same feature. Therefore, it is necessary to clearly define the semantics of the constraint relationships in the new context, where features can have multiple copies, and features can have an attribute type and a value. In this case, we need more expressive approaches to (i) define constraints between features and (ii) perform formal reasoning over the feature models and their constraints.

B. Feature models and their configuration

A configuration of a feature model is usually defined as *the set of features that are selected from a feature model without violating any of the constraints defined in it*, but it can also 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. The first definition, is well-suited when dealing with "traditional" feature models (those that can be defined by using the original FODA notation). In this case, every instantiation of the elements of the feature model will follow the *singleton pattern*, that is, every feature can have at most one instance. Fig. 1 shows an example of this.

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.

As can be seen, this process of selection of features (according to the defined constraints) is closely related with a copy mechanism, that is, a configuration of a feature model is a more restrictive copy of the original one that represents exactly one variant. This mechanism can also be used in cardinality-based feature models. In this case, when features have an upper bound greater than 1 they can be cloned at model level, thus, we can have mutiple *copies* of the same feature. Several specializations can be done at model level until only one variant is possible.

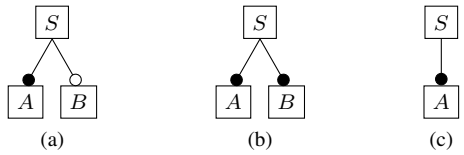


Fig. 1. Example of a feature model (1a) and the two possible configurations that it represents (1b and 1c).

C. Cardinality-based feature models and MDE main issues

In previous paragraphs we have described what cardinality based feature models are and how they are usually exploited. Nevertheless, there are some issues, that we will address in the remaining of this paper, that cause some problems when trying to use feature models in a MDE process.

The first one comes from the classical definition of configuration of a feature model (the set of features that are selected from a feature model). This definition tends to define the configuration as a *copy* mechanism instead of as an *instantiation* mechanism. Although this definition can be somewhat intuitive when dealing with traditional feature models, it can be confusing when dealing with cardinality-based ones. In this case, the instantiation concept is best suited when talking about cardinality-based feature models with attribute types, given that the configuration is more easily understandable as an *instance-of* relationship rather than as a *copy-and-refinement-of* relationship.

The second issue that comes up is how to deal with model constraints when features can be cloned in our models and such features can have an attribute type. In this case, as pointed out in section II, we need to use more expressive languages that allows us (i) to deal with sets of features (i.e. n copies of feature A) and (ii) to deal with typed variables which values can be unbounded in order to easily represent attribute types.

III. Putting Model-Driven Engineering and Feature Modeling together

The Meta Object Facility standard (MOF, [16]), which provides support for meta-modeling, defines a strict classification of software artifacts in a four-layer architecture (from M3 to M0 layer). The meta-metamodel layer (M3) is where the MOF language is found. MOF can be seen as a subset of the UML class diagram, and can be used to define metamodels at M2 layer. This way, artifacts that reside in layer x, are expressed in terms of the constructors defined in layer $x + 1$.

In turn, the Query/Views/Transformations (QVT, [15]) standard describes how to provide support to queries and model transformations in a MDE process. QVT uses the pre-existent *Object Constraint Language* (OCL, [17]) language to perform queries over software artifacts. The QVT standard provides three different languages to describe these transformations. In this sense, the QVT–Relations can be interesting because of its implicit support for traceability and its high level of abstraction, as it is a declarative language. This language uses object templates to define relationships among different domains that can be enforced, performing a model transformation when needed.

As the MOF standard provides support for modeling and metamodeling, we can use it to define cardinality-based feature models by defining its metamodel. Previousy, a configuration

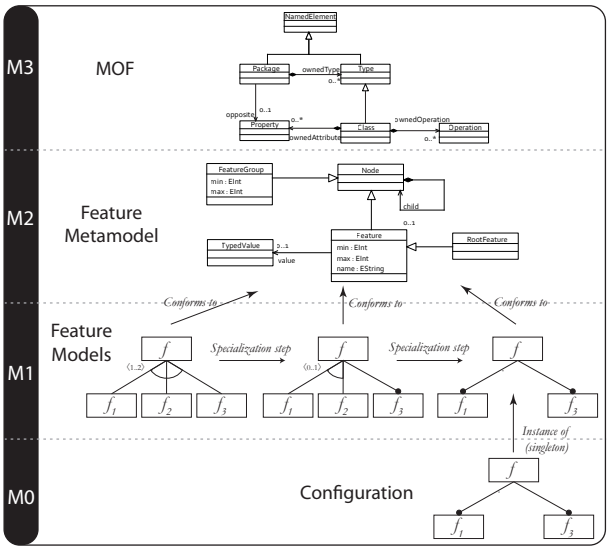


Fig. 2. Specialization and configuration of feature models in the context of MOF

process by means of specialization was shown. This conception about the configuration process involves copy of features, but it has a big implication: configurations are expressed in terms of the feature metamodel instead of in terms of the feature model. Fig. 2 shows how the specialization process fits in the four-layer architecture of MOF. In this figure 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), and finally, in level M1 some feature models are represented. The leftmost model is the one that represents our family of systems, and starting from it, we obtain the final model as an specialization of the original one. Thus, the configuration of the feature model is defined in terms of the features metamodel as the model it conforms to has no variability, and both feature model and configuration are practically equivalent.

In order to use a feature model in a MDE process, we need to use the one that captures the whole variability of the domain. This is necessary to define any possible configuration of the model, but it is also necessary in order to define model-based transformations that allows us to use this feature models and their configurations in other complex processes.

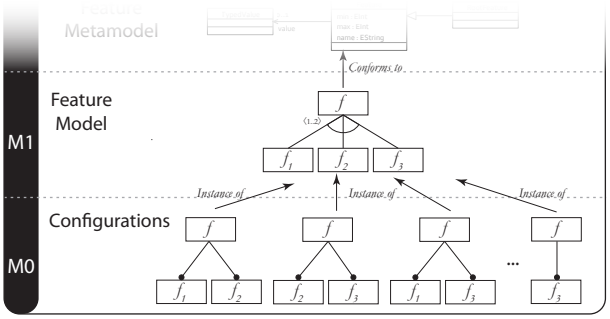


Fig. 3. Instantiation of feature models

Fig. 3 shows how a configuration without performing a specialization process looks like. This way, the feature model can be used in any modeling framework to automatically generate configuration editors, and what is more, feature models and configurations can take part of a MDE process. Developers can take advantage of related tools, feature models can be used to guide model transformations with multiple inputs, and configurations can be automatically checked againts their corresponding models using built-in query languages.

IV. Using a modeling framework for feature models and their configurations

The Eclipse Modeling Framework (EMF) [10] can be considered as an implementations 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 ecore models which will be placed al the metamodel layer (M2). An example of such metamodels can be the metamodel to build cardinality-based feature models. Finally, this 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. 4 shows this architecture.

As can also be seen in Fig. 4, the M0 layer is empty. That is a limitation of most of the modeling frameworks which are available today. As said, EMF provides a modeling language (Ecore) that can be used to define new models and their instances. This approach 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. 4 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 Varibility Model, DVM, which represents the Feature model as a new class diagram). Thus, it is possible to represent a feature model at the metamodeling layer, making

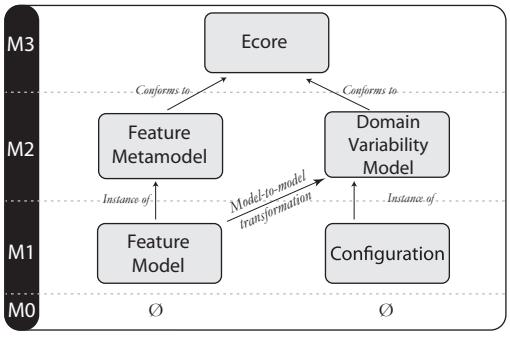
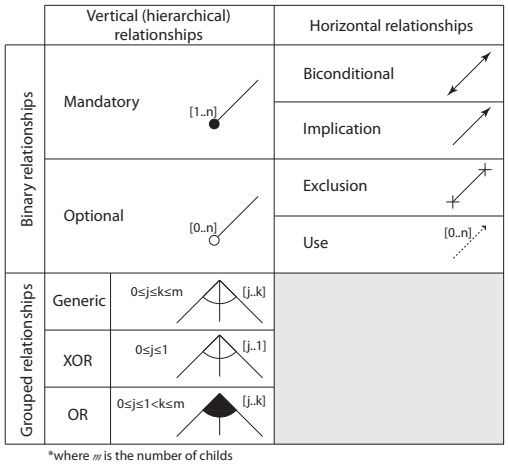


Fig. 4. EMF and the four-layer architecture of MOF

TABLE I
CARDINALITY-BASED FEATURE METAMODEL: PROPOSED TYPES OF RELATIONSHIPS BETWEEN FEATURES



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.

V. 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.
- Automatic support to generate Domain Variability Models from feature models that capture all the variability of the application domain, allowing the developers to use them in model transformations.
- Automatic support for configuration editors, which will assist the developers in the task of defining new configurations.
- Capabilities to check the consistency of a configuration against its corresponding feature model.

A. Cardinality-based feature metamodel

The basis of our work is the cardinality-based feature metamodel, which permits to define feature models. In our proposal we have decided to represent explicitly the relationships between features. Thus, our metamodel represents in an uniform way the hierarchical relationships and the restrictions between features. Table I classifies and summarizes the types of relationships that the feature metamodel is able to represent. As can be seen, relationships are classified in two orthogonal groups:

- *Vertical vs. horizontal relationships.* Vertical relationships define the hierarchical structure of a feature model and horizontal relationships define dependecies and restrictions between features.
- *Binary vs grouped relationships.* Binary relationships define relationships between two single features. In turn, grouped relationships are a set of relationships between a single feature and a group of childs.

Given this classification, the following relationships exist:

- *Binary and vertical relationships*. This relationships define structural relationships between two single features. In our approach, they represent a *has_a* relationship between a parent and a child feature. They can be mandatory and optional depending on the lower bound value. The upper bound (n) can be on both cases 1 or greater than 1, and indicates how many instances of the child feature will be allowed.
- *Grouped and vertical relationships*. Grouped and vertical relationships are a set of binary relationships where the child features share a *is_a* connotation with respect to their parent feature. A group can have an upper and a lower bound. These bounds specify the minimun and the maximun number of features that can be instantiated (regardless of the total number of instances).
- *Binary and horizontal relationships*. These relationships are specified between two features and do not express any hierarchical information. They can express constraints (biconditional, 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* ($A \nleftrightarrow B$): If an instance of feature A exists, can not exist any instance of feature B and vice versa.
  - *Use* ($A \dashrightarrow B$): 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).

Fig. 5 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. In that figure, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features and the set of Relationships among them. A feature model must also have a root feature, which is denoted by means of the rootFeature role.

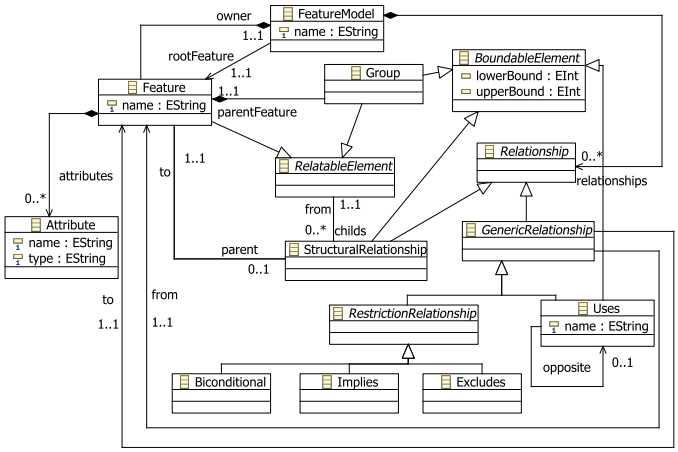


Fig. 5. Cardinality-based features metamodel

Binary relationships in table I are represented in the features metamodel as descendants of the Relationship class. Class StructuralRelationship represents the so called *Vertical* relationships and GenericRelationship represents the *Horizontal* ones. StructuralRelationships relate one parent RelatableElement (a Feature or a Group) with one child Feature. A Group specifies that a set of StructuralRelationships should be considered as a group.

It is noteworthy to point out two slight differences of our approach with respect to the classical cardinality-based feature models. First, we represent feature multiplicities at relationship level instead of at feature level (by means of the BoundableElement class). This allows us to easily define mandatory and optional relationships explicitly. Second, features can not have an attribute type. In turn, this information is expressed in terms of feature attributes. Feature attributes express information which is complementary to a feature and can be used to describe *parametric features*.

B. Cardinality-based feature modeling editor

The cardinality-based feature modeling editor allows us to easily define new feature models. Following the Model-Driven Software Development (MDSD) approach, it has been automatically generated from the metamodel presented in the previous section. To obtain this graphical editor, the Graphical Modeling Framework (GMF [9]) has been used.

Fig. 6 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 canvas an example feature model is shown. This feature model describes a product line for mobile phones. A mobile phone must have a screen, which can be touchscreen or not. Touchscreens can use resistive or capacitive technology. Moreover, a mobile phone can also support handwriting recognition, however, this feature is incompatible with normal screens. This can be described by means of an excludes relationship (solid line with crosses at its ends between feature Normal and HandwrtingRecognition). In our product line, mobile phones can also have one or two cameras, each one with their corresponding resolutions. Those cameras can be used for videoconferencing or digital photography. If the camera is used for digital photography, it can be associated to an optional flash light. This is specified by means of the dashed line between features Camera and Flash. Finally, these mobile phones can have FM radio support, but it can be installed only if the device has a headphones connection, as the FM antenna is part of this accessory. This dependency is expressed by the implies relationship (directed line between feature Radio and feature Headphones).

C. Generating the Domain Variability Model

In section IV was explained that it is necessary to execute a model-to-model transformation in order to easily define configurations of a feature model in EMF. Following the MDA approach, this transformation has been defined by using the Relations language defined in the QVT standard. In order to integrate and execute the transformation process in our

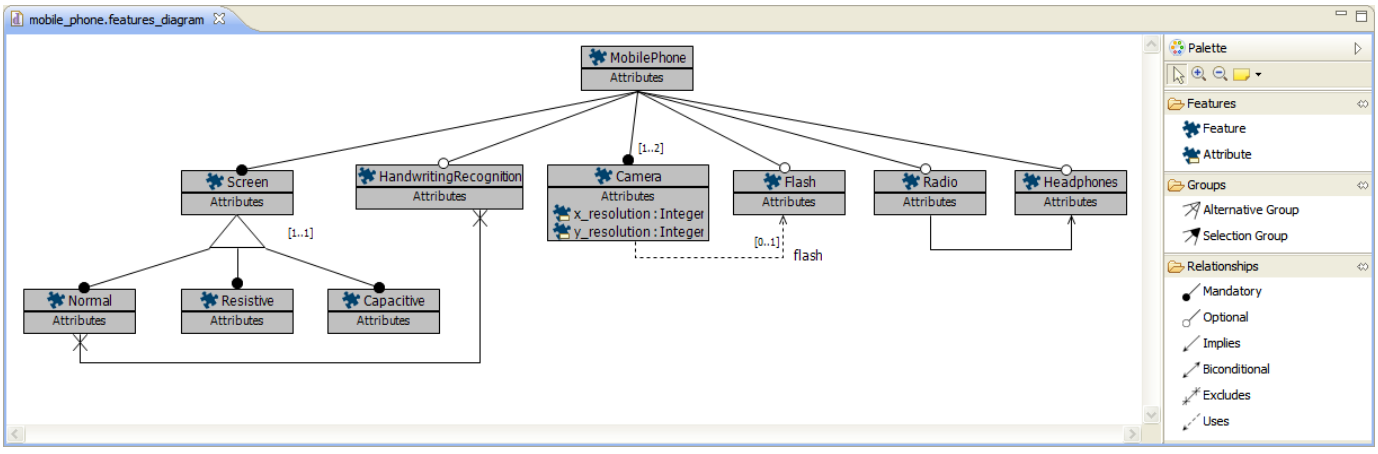


Fig. 6. Screenshot of the cardinality-based feature modeling editor

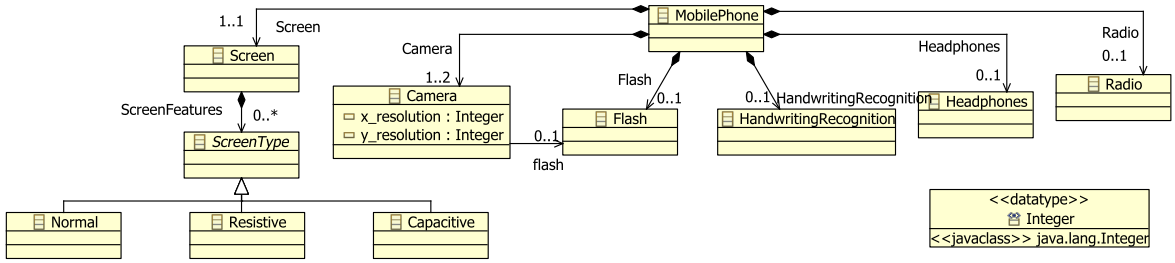


Fig. 7. Example Domain Variability Model

prototype, a custom tool based on the mediniQVT [11] transformations engine has been built.

The following paragraphs describe the transformation which transforms a feature model (expressed as an instance of the cardinality-based features metamodel in EMF), to a Domain Variability Model (DVM, expressed as an Ecore model that can be instantiated). The transformation is declared as follows and it is executed from the *feature* domain to the *classdiagram* domain.

```
transformation Feature2ClassDiagram(feature :
    features, classdiagram : ecore) { ... }
```

As feature models describe not only the structure of the features but also the relationships among them, it is necessary to define rules to generate both the structure of the DVM and the restrictions that apply to it. First, the structure of the DVM is defined by means of Ecore containment references and inheritance relationships; and second, restrictions are defined by means of OCL expressions. These OCL expressions are included on the DVM itself by means of *EAnnotations*. This *EAnnotations* are automatically used in next steps by our prototype to check that configurations are valid.

Figure 7 shows the resulting Ecore model. The rules that have been applied are:

- *Feature2Class*. For each Feature of the source model an *EClass* with the same name of the feature will be created. All these classes will be created inside the same EPackage, whose name and identifier derives frome the feature model name. All the features in Fig. 7 are example of the aplication of this rule.
- *FeatureAttribute2ClassAttribute*. For each feature Attribute, an EAttribute will be created in the target model. This EAttribute will be contained in its corresponding EClass. Any needed EDataType will be also created. Attributes in the Camera EClass are example of this.
- *StructuralRelationship2Reference*. For each StructuralRelationship from a parent Feature a *containment* EReference will be created from the corresponding EClass (i.e. same name than the Feature). The multiplicity of this EReference will be the lower and upper bounds of the StructuralRelationship. *Containment* EReferences from MobilePhone are example of the application of the rule.
- *Group2Reference*. This rule states that for each Group contained in a Feature a *containment* EReference will be created from the corresponding EClass (i.e., same name than the Feature). This EReference will point to a new abstract class, whose name will be composed by the Feature name and the suffix "Type". This rule has the following post-conditions: *GroupChild2Class*, *Group2ChildsAnnot*, *GroupChild2LowerAnnot* and *GroupChild2UpperAnnot*. Screen, ScreenType and the EReference ScreenFeatures are example of the result produced by this rule.
- *GroupChild2Class*. This rule is in charge of creating the EClasses from the Features belonging to a Group. Moreover, each one of these EClasses inherit from the abstract *EClass* that has been previously created. EClasses Normal, Resistive, Capacitive, and their inheritance relationship are example of the aplication of the rule.
- *Group2ChildsAnnot*, *GroupChild2LowerAnnot*, and *GroupChild2UpperAnnot*. These rules create EAnnotations that will contain OCL expressions. This expressions will check that the multiplicities specified for the Group and the child Features are also satisfied by the instances of the DVM.
- *UsesRelationship2Reference*. 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. The EReference flash, between Camera and Flash, shows and example of this.

The next relations generate OCL expressions in the DVM for each restriction relationship of the source model. They can be easily expressed as the following OCL invariants created in the root EPackage:

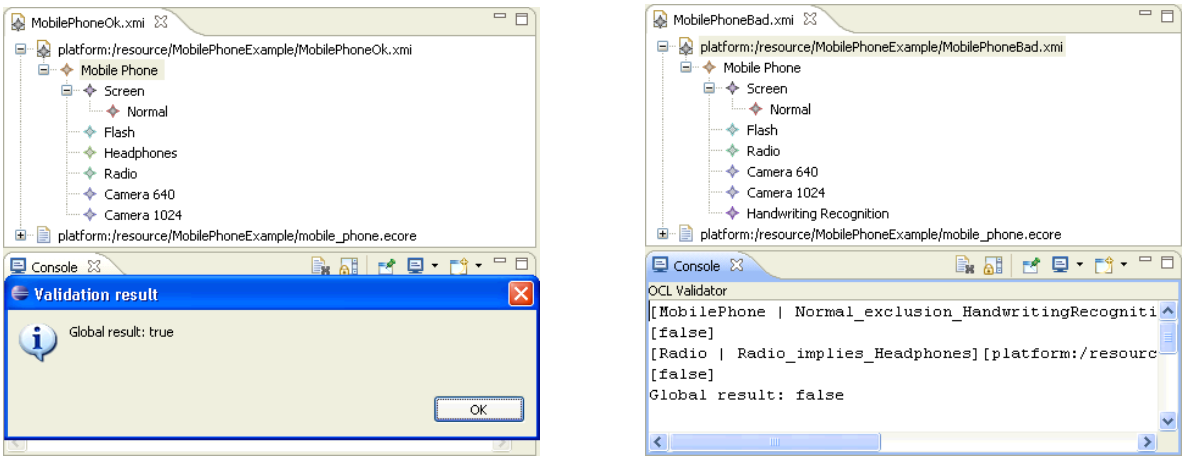
- *ExcludesRelationship2ModelConstraint*. This rule generates the following invariant for (`A excludes B`):

```
A.allInstances()->notEmpty() implies B.
    allInstances()->isEmpty()) and
B.allInstances()->notEmpty() implies A.
    allInstances()->isEmpty())
```

- *ImpliesRelationship2ModelConstraint*. If the relationship is (`A implies B`), the following OCL expresion is created by this rule:

```
A.allInstances()->notEmpty() implies B.
    allInstances()->notEmpty())
```

- *BiconditionalRelationship2ModelConstraint*. This rule creates the following OCL invariant if the source relationship is (`A if and only if B`):



(a) (b)

Fig. 8. Example of successful (8a) and unsuccessful (8b) configuration check

```
A.allInstances()->notEmpty() implies B.
    allInstances()->notEmpty()) and
B.allInstances()->notEmpty() implies A.
    allInstances()->notEmpty())
```

D. 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 varibility than the original feature model, developers can use the standard Ecore tools. The most straightforward method to create a new configuration is to use the "Create Dynamic Instance…" option of the standard "Sample Ecore Model Editor". This way, the "Sample Reflective Ecore Model Editor" can be used to define new configurations of our feature model. However, although EMF provides all the libraries and technologies to exploit OCL expressions, there is not a default method 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. 8a shows an example configuration. A mobile phone with a normal screen, radio, headphones and two cameras has been defined. This configuration is valid conforming to the example feature model, as the popup window in the figure shows. However, Fig. 8b shows a configuration with a normal screen and handwriting recongnition, which violates the excludes relationsship. Moreover, this configuration includes radio support, but it does not include the headphones connection which is also invalid. 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.

VI. Related works

Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals for variability management have arised. Specially, most of them are based in the original FODA notation and propose several extensions to it [5]. Our work is closely related with previous research in feature modeling, however, there are several distinctive aspects:

In [8] a notation for cardinality-based feature modeling is proposed. In this sense, our tool shares most of this notation as it is widely known and used, but we have included some variants. First, in our aproach features can not have an *attribute type*, but rather, they can have typed *feature attributes* which can be used to describe *parameterized features*. Second, in [8] both feature groups and grouped features can have cardinalities. However, the possible values for grouped features cardinalities are restricted. In our proposal, these values are not restricted and have different meanings: cardinality of feature groups specify the number of features that can be instantiated, and cardinality of grouped features specify the number of instances that each feature can have.

Our work describes a prototype to define configurations of feature models. Previous work has been also done in this area, such as the *Feature Modeling Plugin* [1]. This tool allows the user to define and refine a feature model and configurations by means of specializations. The advantage of this approach is that it is possible to guide the configuration process by means of constraint propagation techniques. 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 process is properly done).

Some previous works have already represented feature models as class diagrams. In [8] the translation from feature models to class models is performed manually, and no set of transformation rules are described. In this work, OCL is also presented as a suitable approach to define model constraints, but as the correspondences between feature models and class diagrams are not precisely defined, there is no automatic

generation of OCL invariants. In turn, [13] do present a set of QVT rules to automatically generate class diagrams from feature models. 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. Specifically, this work proposes to represent features as propositions and restrictions among them as propositional formulas. However, in propositional formulas only `true` and `false` values are allowed. This approach is not suitable to our work, as we can have typed attributes which can not be expresed by this kind of formulas. Thus, we state that more expressive languages are needed. In this case, we propose OCL as our constraint definition language. Nevertheless, in order to perform more advanced reasoning (for example, satisfiability of feature models) richer formalisms are needed.

VII. Conclusions

In this paper we have proposed a prototype to define and use feature models in a MDE process. This prototype addresses one of the main issues that arises when dealing with nowadays metamodeling tools: they usually are not able to deal simultaneously with artifacts located in all the MOF layers. For example, the selected modeling framework (EMF) is only able to represent three different MOF layers (from M3 to M1), so that, our prototype defines a mechanism to overcome this typical limitation by means of model transformations. This way, feature models can be transformed to Domain Variability Models that can be instantiated and reused in future steps of the MDE process.

Our tool has been also designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. Thus, 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.

It is important to remark that having both feature models and configurations at different layers is very useful as they can easily be used in model transformations. In [3] an example is shown. In this work, a model transformation with multiple inputs is used to generate a software architecture automatically. In this case, one of the arguments of this transformation is a configuration of a feature model, which is used to guide the architecture generation process.

Finally, to use DVMs allows us to address some satisfiability problems from new points of view. The introduction of cardinalities and unbounded attribute types makes harder to reason about feature models. Thus, richer formalisms (compared with the traditional ones) are needed. Fortunately, class diagrams are widely used and known, and several formalisms to reason about them have been proposed. In this sense, we have already done some preliminary works in model consistency checking by using formal tools [4], and future works are oriented towards this direction. A first version of our prototype can be downloaded from `http://issi.dsic.upv.es/~agomez/feature-modeling`.

Acknowledgments

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] J. Cabot, R. Clarisó, and D. Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548, NY, USA, 2007. ACM.

[5] 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.

[6] P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[7] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[8] K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.

[9] Eclipse Organization. The Graphical Modeling Framework, 2006. http://www.eclipse.org/gmf/.

[10] EMF. http://download.eclipse.org/tools/emf/scripts/home.php.

[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 2.0 QVT final adopted specification (ptc/05-11-01). 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[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.

[17] Object Management Group. OCL 2.0 Specification. 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01.