University of Waterloo, ECE, Technical Report No. 2008-08

Matthew Stephan · **Michał Antkiewicz**

Ecore.fmp

A tool for editing and instantiating class models as feature models

Version 1, last update: May 30, 2008

Abstract Ecore Feature Modeling Plug-in (Ecore.fmp) is a tool for editing and instantiating class models as feature models. The tool interprets a class model as a feature model and an object model as a feature configuration, thus exposing the variability aspect of class models. Furthermore, the tool supports instantaneous synchronization between class models and feature models so that the changes made to one model are propagated to the other model. In this report, we describe a bi-directional mapping between class and object models to feature models and configurations that is needed for synchronization. We argue that both class modeling and cardinality-based feature modeling have similar expressive power and we discuss some issues related to the semantic mismatch between the two notations.

Keywords class modeling · cardinality-based feature modeling · Ecore · configuration · instantiation · variability

1 Introduction

Many class modeling tools, such as MagicDraw [13], Microsoft Visio [12], and Rational Software Modeler [10], emphasize a graph-like view in which classes are represented as nodes and relationships among classes are represented as edges. While this default graph-like view is sufficient for general modeling, it does not offer specialized views of the class model because none of the relationships among classes are dominant. For example, choosing inheritance as a dominant relationship and illustrating an inheritance hierarchy requires manual rearrangement of the classes. Also, the variability aspect of the class model is not easily visible, that is, it is not immediately apparent which classes must be or may optionally be instantiated and which attributes must be or may optionally be set.

Matthew Stephan
University of Waterloo E-mail: mdstepha@uwaterloo.ca

Michał Antkiewicz
University of Waterloo E-mail: mantkiew@uwaterloo.ca

Feature modeling is a technique and a notation specialized for variability modeling. A feature model consists of a hierarchy of features that are properties of their parent features. Feature modeling supports variability modeling by offering constructs such as optional features and alternative (XOR) feature groups.

In this report, we focus on two concrete notations for class modeling and feature modeling: *Ecore* [2] and *cardinality-based feature modeling* [8], respectively. Ecore is a simple class modeling notation offering constructs such as packages, classes, attributes, references, and annotations. Cardinality-based feature modeling offers constructs such as features, feature attributes, and feature groups. In cardinality-based feature modeling, features and feature groups have a *cardinality* that specifies the kind of a feature or a feature group. For example, a feature with the cardinality [0..1] is an optional feature and a feature group with the cardinality [1..1] is an XOR feature group. Semantically, a class model describes a set of valid *object models* where the objects are instances of the classes. Similarly, a feature model describes a set of valid *feature configurations* that contain instances of features from the model. We refer to the creation of object models for a given class model as *instantiation*. We refer to the creation of a feature configuration for a given feature model as *configuration*.

In this report, we propose an approach to viewing, editing, and instantiating class models as feature models. We implemented the approach in a feature modeling and configuration tool called Ecore.fmp (Ecore Feature Modeling Plug-in). Ecore.fmp consists of two Eclipse views: *feature modeling view*, for viewing and editing a class model expressed in Ecore as a feature model, and *feature configuration view*, for instantiating an object model using feature configuration. Users of Ecore.fmp can modify their class models in both the default Ecore editor and the feature modeling view simultaneously and independently. Similarly, instantiation of class models can be accomplished through the Ecore object model editor or the feature configuration view. It is important to note that a graphical editor for Ecore class models that uses a standard graph-like notation is also available and can be used in conjuncion with Ecore.fmp.

Ecore is the metamodel that every class model conforms to. In order to facilitate synchronization between a class model and a feature model as well as between an object model and a feature configuration, we defined a specialized metamodel for feature models and configurations. The specialized metamodel, called *Ecore.fm*, has references to Ecore model elements which represent feature model and configuration elements. In this technical report we present the metamodel for feature models and configurations as well as a bi-directional mapping between that metamodel and Ecore. The bi-directional mapping is implemented in Ecore.fmp in the form of operations on feature models or configurations that immediately modify Ecore models accordingly. Analogously, modifications on Ecore models are interpreted as modifications on feature models or configurations.

Furthermore, we discuss some issues related to the semantic mismatch between feature models and class models. Information on the design and implementation of the current prototype is also provided. The report gives enough background information on class modeling and feature modeling such that readers with a minimal knowledge of both will be able to understand the contents of this report. The report concludes with a discussion of related work and possible future work in this direction.

2 Ecore

Ecore is a class modeling notation equivalent to the essential subset of the Meta-Object Facility (MOF), the Essential MOF. MOF is an Object Management Group (OMG) standard for expressing metamodels of modeling languages such as UML. Its use is quite prevalent in industry today, coming in three different variants, Complete, Semantic, and Essential MOF [3].

Ecore is a part of the Eclipse Modeling Framework (EMF), a framework that provides a practical foundation for building modeling tools. When using EMF, users typically first create a metamodel for their domain in the form of an Ecore class model. Next, the users may use a code generator that generates an implementation of the metamodel or they can utilize the metamodel using reflection. The code generator can also create a specialized graphical tree editor for creating and editing object models for the given metamodel.

Analogously to MOF, which is itself defined using MOF, Ecore is also defined using Ecore, that is, the metamodel of Ecore is also a class model that conforms to Ecore. A fragment of the simplified metamodel of Ecore is presented in Figure 1.

Every class model conforming to Ecore consists of an instance of the class `EPackage` (package), which can contain instances of the subclasses the class `EClassifier`: `EClass` (classes), `EDataType` (primitive and user defined types), and `EEnum` (enumerations). Instances of the class `EClass` contain instances of the subclasses of `EStructuralFeature`: `EAttribute` (attributes) and `EReference` (references). Instances of the class `EReference` point to instances of the class `EClass` through the `eReferenceType` reference. The attribute `containment` of the class `EReference` indicates whether the reference is a containment reference. Ecore, similarly to Essential MOF, also supports a tagging mechanism which allows annotating each class model element with a set of key-value pairs. In Ecore, instances of the class `EModelElement` can contain many instances of the class `EAnnotation`, which, in turn, has a String to String map that represents key to value pairs. The complete metamodel of Ecore can be found at [1].

The Ecore metamodel shown in Figure 1 is also an example class model that conforms to Ecore. The class model is shown in a standard UML-like notation, in which classes (instances of the `EClass` class) are shown as boxes, attributes (instances of the `EAttribute` class) are listed inside classes, and references (instances of the `EReference` class) are shown as arrows. Note that the inheritance arrows link classes with the values of these classes' `eSuperTypes` reference.

3 Class Modeling vs. Feature Modeling

In object-oriented analysis and design, class modeling plays an important role as it allows decomposing a software system into classes, assigning responsibilities to those classes, and modeling various kinds of relationships among the classes.

Feature modeling, on the other hand, emphasises modeling of the system in terms of its properties, that is, the system's features. In feature modeling, the system is decomposed into a hierarchy of features that directly or indirectly describe it. Feature modeling has its roots in domain analysis [11], where it is used for characterizing domain concepts by their features. More recently, feature modeling is used in modeling software product lines thanks to its support for variability modeling.

However, despite those different uses of class modeling and feature modeling, both notations have very similar expressive power. In fact, cardinality-based feature modeling with inheritance and reference attributes is equivalent to the part of class modeling for structural modeling, which we show in this technical report.

Figure 2 shows a feature model that represents the class model from Figure 1. A feature hierarchy is rendered as a tree in which subfeatures are further right. Table 1 summarizes the cardinality-based feature modeling notation we use in this report. The feature `EPackage` is a *root feature* and corresponds to a class that has been designated as a root. Root features are also classes that are not contained through any containment reference. The feature `EClass` corresponds to a containment reference of the class `EPackage`, which contains instances of the class `EClass`. Similarly, the feature `EStructuralFeature` corresponds to the containment reference `EStructuralFeature` of the class `EClass`. The features `EAttribute` and `EReference` correspond to the concrete subclasses of the class `EStructuralFeature` that can be instantiated. Furthermore, the cardinalities of the features correspond to multiplicities of references and

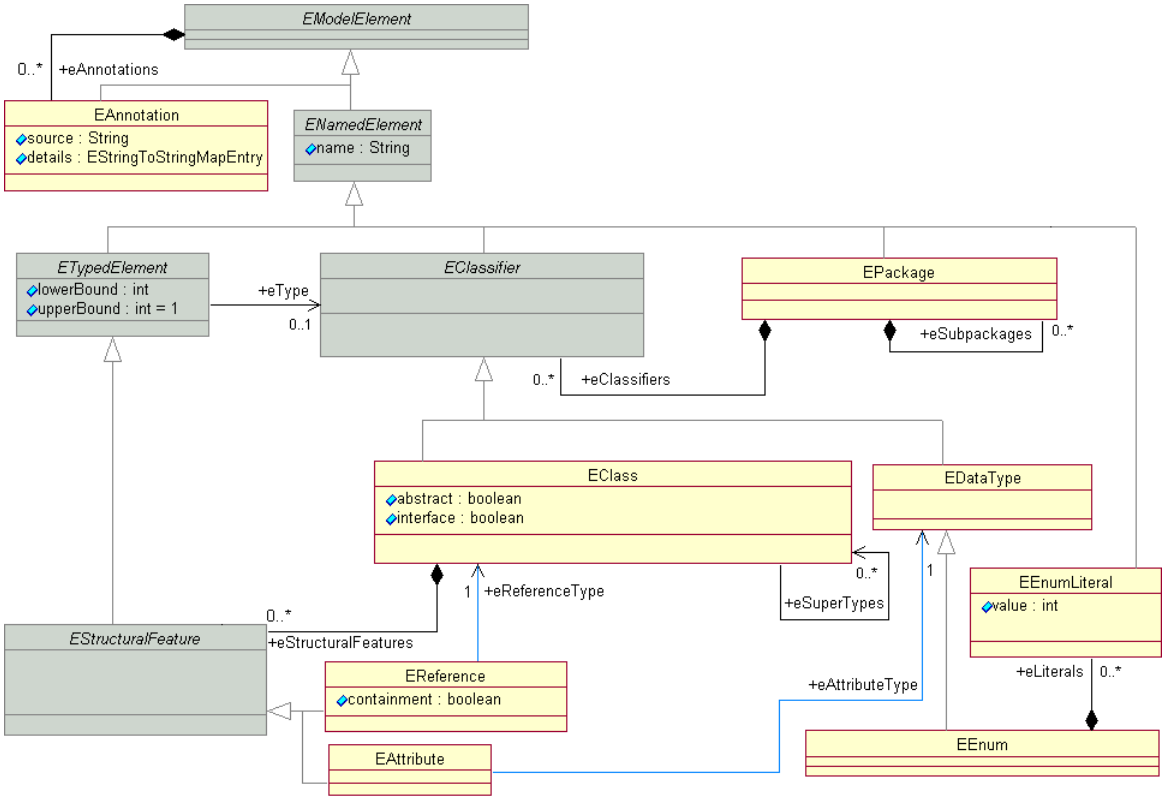


Fig. 1 A fragment of the metamodel of Ecore

Table 1 Cardinality-based feature modeling notation

icon	feature model or configuration element
	package
	root feature
	optional feature [0..1]
	mandatory feature [1..1]
	optional multiple feature [0..*]
[0..m]	optional multiple feature [0..m]
	mandatory multiple feature [1..*]
[n..m]	mandatory multiple feature [n..m], $n > 0$
--|>	inheritance
	feature group <1-1> (XOR)
<n..m>	feature group <n..m>
	grouped feature [0..1]
	selected optional feature
	missing mandatory feature
	missing mandatory multiple feature

attributes. For example, the feature `eReferenceType` belonging to feature `EReference` is a mandatory feature with cardinality [1..1] (indicated by a filled circle) because the multiplicity of the attribute `eReferenceType` is [1..1]. Exceptions to this rule are boolean attributes which are interpreted as optional features (e.g., `containment` belonging to `EReference`), in which case, the value `true` of the attribute indicates the presence of the feature. Features can correspond to multiple elements from the class model, for example, the feature `EClass` belonging to `EPackage` corresponds to both the containment reference `eClasses` and the class `EClass`. Also, elements from the class model can correspond to multiple features, e.g., if a certain class is the type of multiple containment references.

In this report, we define the bi-directional mapping between class models and feature models that can be used for interpreting class models as feature models and vice versa. First, we need to define a metamodel for representing feature models and configurations: Ecore.fm.

4 Ecore.fm: a Metamodel of Feature Models and Configurations

The Ecore.fmp is built upon an Ecore model itself in order to utilize the capabilities provided in EMF, such as code generation and validation but also to support direct references from feature model elements to their corresponding Ecore

Table 3 Elements within Ecore FMP Node Class

Element Name	Description
min	Attribute that represents the lower bound of the cardinality of the node. There must be at least this many instances of what this node represents in a correct feature configuration.
max	Attribute that represents the upper bound of the cardinality of the node. There may be at most this many instances of what this node represents in a correct feature configuration. Similarly to a convention in Ecore, the value -1 indicates an infinite (*) upper bound.
children	Reference that containes the set of children nodes of the given node in a feature model and configuration.
ecoreClass	Reference that points at a class in an Ecore model that represents the given node. This reference must be set.
ecoreStructuralFeature	Reference that points at a structural feature (an attribute or a reference) that represents the given node. This reference may not be set.
ecoreContainingFeature	Reference that points at a containment reference in an Ecore model that contains the class representing the given node. It is used for mapping back to Ecore and other operations. This reference may not be set.
ecoreInstance	Reference that points at an object (instance of an EClass) that represents the given node. This value is only used in feature configuration when a feature has been selected. This reference may not be set.

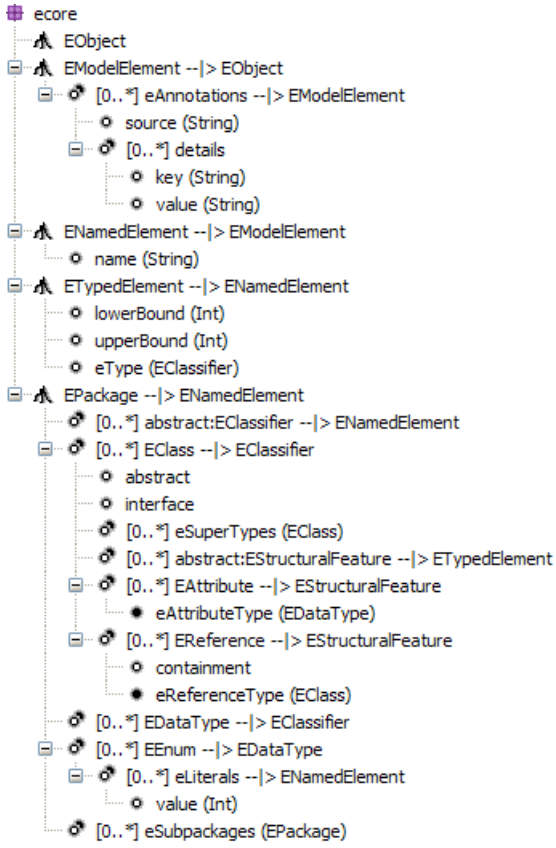


Fig. 2 A fragment of the metamodel of Ecore as a feature model

model elements. Therefore, the Ecore.fm metamodel is a class model expressed in Ecore. In the remainder of this report, we use a more compact notation for class models: the notation used by the default EMF Ecore class model editor. We modify the notation by introducing new icons for abstract classes and containment references in order to visually distinguish them from concrete classes and non-containment references. Table 2 summarizes the notation used for Ecore class models.

Figure 3 shows the Ecore.fm metamodel. `Node`, `Feature`, and `FeatureGroup` are classes; `FeatureRepresentation` and `GroupRepresentation` are enumerations; `min`, `max`, and `representation` are attributes; `children` is a containment reference; and `ecore*` are non-containment references. The basis for the metamodel is the class `Node`, which is an abstract class that represents any entry in either a feature model or a feature configuration view. The class `Node` contains other nodes throught the reference `children`. The class `Feature` is a subclass of `Node` and represents a solitary or a grouped feature within a feature model. Analogously, the class `FeatureGroup`, a subclass of `Node`, represents a feature group. The only notable difference between features and feature groups, with respect to the metamodel, is the enumeration attribute `representation`. The enumeration `FeatureRepresentation` has five literals, each describing a combination of class model elements a feature corresponds to. The enumeration `GroupRepresentation` has two literals describing the type of the feature group. Table 3 provides a brief description of the attributes and references of the class `Node`.

Table 2 Ecore class modeling notation

icon	Ecore class model element
	package
	abstract class
	concrete class
->	inheritance
	attribute
	reference
	containment reference
1 , 1..*	mandatory multiplicities
, 0..*	optional multiplicities
	enumeration
	enumeration literal
	annotation
	annotation detail (key-value pair)

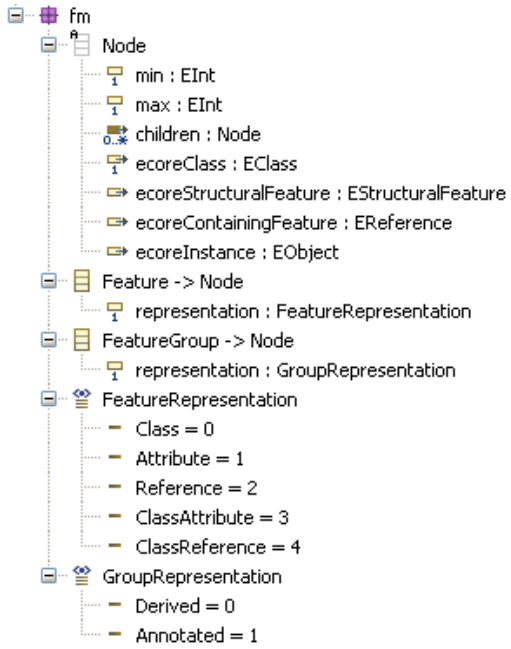


Fig. 3 Ecore.fmp Meta Model

The following subsections we describe each direction of the bidirectional mapping between Ecore and Ecore.fm metamodels, as outlined in Figure 4. The *class to feature* direction covers the mapping of Ecore class and object model elements into feature model and configuration elements, respectively. The other direction, *feature to class*, is the inverse mapping. This two-way mapping is necessary for being able to propagate changes made in one model to the other one and thus maintain consistency between the models at all times.

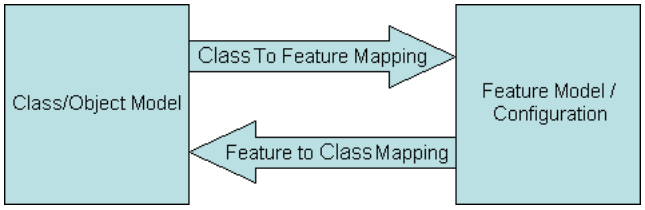


Fig. 4 Mapping Between Class Model and Feature Model

4.1 Class to Feature Mapping

The class to feature mapping involves interpreting an Ecore class model and rendering it as a feature model that is semantically equivalent. Ecore elements such as classes, attributes, and non-containment references are represented as features and annotated classes and containment references to classes involved in inheritance are represented as feature groups.

The interpretation of a class model as a feature model begins by determining root features. Root features are represented as classes that are not directly or indirectly contained (referenced to through a containment reference) by other classes. A class is indirectly contained if any of its superclasses is contained. In the case of *cyclic containment* whereby each class is contained, a root feature can also be manually specified by annotating a class with the annotation `root`. Specifically, an `EAnnotation` that has its attribute `source` set to `root` must be added to the class. If all classes are contained and none of them is specified as a root, the resulting feature model is empty.

For each root feature, the children of the root feature are then added based on the structural features of the classes that represent the root features. The structural features are interpreted as features or feature groups and the values of the attribute `representation` of the `node` are set accordingly. First, it is ascertained whether the node is a solitary or a grouped feature by looking at the nature of the structural feature. Table 4 presents the cases in which a node is a solitary feature. Feature groups are created for an Ecore class that is annotated with `featureGroup` or for a containment reference with a finite upper bound that points to an Ecore class that has subclasses. We elborate on feature groups in Section 6.3. Once the type of node is discovered, it must be further classified into the appropriate feature or feature group representations. Table 4 discusses the five types of representation of features and the way they are identified.

For each feature that is of type `Class`, `ClassAttribute`, or `ClassReference`, their children/subfeatures are realized by iterating through the structural features of the corresponding class that the feature represents the same way as discussed above for a root feature. This process continues until there are no more children to discover, that is, all the features have either no children or have children that are already accounted for. In the case of cyclical containment, the infinite recursion is terminated by presenting a containment reference as a feature with a reference attribute (e.g., the feature `eSubpackages` in Fig. 2), which is not further expanded.

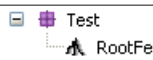
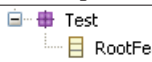
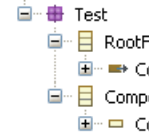
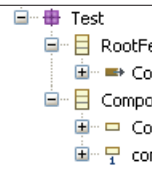
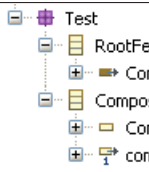
4.2 Feature to Class Mapping

The feature to class mapping is required in instances where the user wants to modify the Ecore model by executing commands via the feature model. Adding, removing, and modifying features in the feature model trigger the appropriate transformations to be executed on the Ecore model. Table 5 discusses the mapping by providing a description and an image for the feature model and the corresponding Ecore model. For the purpose of this report, an atomic feature is a feature with no children while a composite feature is a feature with one or more children. These cases form the basis of the feature to Ecore mapping. Cardinality of attributes and references are handled by obtaining the lower and upper bounds in the feature model and using them when constructing the Ecore attribute or references.

Table 4 Feature Types and Essential Characteristics

Feature Repres.	Essential Characteristics	Description
Class	Is either a root feature or an Ecore class that is contained via an Ecore containment reference. The Ecore class must not own a structural feature with the same name (case insensitive) as the class.	Represents a standard class that does not have an identifying / special attribute or reference.
Attribute	Is an Ecore Attribute structural feature that is contained by a class.	Represents an attribute belonging to a class.
Reference	Is typically a non-containment reference to another Ecore class. In some instances, it may also represent a reference to a contained class that is contained higher in the feature's hierarchy (cyclic containment).	Represents a reference to a class that is not contained. In case of cyclic containment, this represents a reference to a contained class whose contents can not be shown because it would continue on indefinitely. In both cases, children should not be shown.
Class Attribute	Is a contained Ecore class that owns an Ecore attribute as one of its structural features that has the same name (case insensitive) as the class. For example, Class "Person" that has an attribute named "person".	This is a special type of Class feature that has a defining attribute. It can be thought of as a typed feature that has children.
Class Reference	Is a contained Ecore class that owns an EReference as one of its structural features that has the same name (case insensitive) as the class. For example, Class "Animal" that has an reference named "animal".	This is a special type of Class feature that has a defining reference.

Table 5 Feature to Ecore Mapping

Description	Feature Model	Ecore Representation
A Root Feature with no type is represented as a standard class in Ecore.		
An atomic feature that has no type is represented as an EBoolean.		
An atomic feature that has a primitive type is represented as an Ecore Attribute with the corresponding Ecore type.		
An atomic feature that has a type that is referring to a (non-primitive) feature is represented as an Ecore reference.		
A composite feature that has no type is represented as a standard Ecore class.		
A composite feature that has a primitive type associated with it is represented as a feature of type Class Attribute. Thus, it is given an attribute with the same name as the class that has a 1..1 cardinality constraint.		
A composite feature that has a reference to a non-primitive type in the feature model associated with it is represented as a feature of type Class Reference. Thus, it is given a reference with the same name as the class that has a 1..1 cardinality constraint.		

5 Ecore.fmp Feature Configuration

Feature modeling corresponds to class modeling, that is, a feature model corresponds to a class model. A given feature model describes a set of valid feature configurations. Analogously, a class model describes a set of valid object models. When a feature is present in a feature configuration, it means that the object of the corresponding class must be present in the object model and the value of the corresponding attribute or reference must be set for the corresponding object. Similarly, features corresponding to existing objects and values of attributes and references in an object model must be present in the feature configuration.

Just as the feature modeling view of Ecore.fmp allows for simultaneous editing of the feature model and the class model, the feature configuration view allows for simultaneous editing of the feature configuration and the object model. The feature configuration functionality of the plug-in also uses the Ecore.fm metamodel described in Section 4; however, feature configurations are presented differently from feature models. Each element in a feature configuration has children that are comprised of elements that correspond to existing elements in the object model as well as elements from the feature model for which elements in the object model have not been created yet. Displaying elements for non-existing object model elements allows the user to see which configuration steps are possible at any given time during configuration and is the usual way of performing configuration. Another difference between feature modeling and feature configuration is related to abstract and prohibited (with cardinality [0..0]) features, which are not shown in feature configuration because it is incorrect for a user to be able to instantiate such elements. However, if an instance of such an element exists, it will be shown as an error. Furthermore, all features inherited from abstract features are shown as children of the features that inherit them because the user must be able to configure those features (otherwise they would not be visible because abstract features are not shown). Another important difference is that features for non-existing object model element are shown only if the user can actually configure them. These features that are able to be configured can be considered prototypes, as done in [4]. Prototypes are a feature from the feature model for which object model elements do not exist but are able to be cloned/created. For example, if an upper bound of a multiple feature has been reached, the prototype can no longer be cloned and therefore it is not shown in a feature configuration.

To summarize, the algorithm for determining the children for a node in a feature configuration first displays elements from a feature model depicting all feasible prototypes that can be created and then it displays children corresponding to existing object model elements. After performing a configuration step, the corresponding object model is modified using EMF reflection: objects are created for classes and values are set for attributes and references. Reflection is also used during the interpretation of the object model as a feature configuration.

Lastly, images in feature configuration vary depending on the cardinalities of the feature and their parents. Similarly to fmp [4], optional features are represented and modified through a check box and multiple features' prototypes can be cloned by double clicking. For features with representation type `Attribute`, `Reference`, `ClassAttribute`, and `ClassReference`, instantiation triggers a dialog that requests the appropriate value (primitive value for attributes and EObject for references) from the user. When cloning a feature with representation `Class` belonging to a containment reference, a new EObject is created with the feature's defining attributes/references specified by the user if the feature is represented as `ClassReference` or `ClassAttribute`. When cloning a feature with representation `Class` that is referred to by a non-containment reference, the user is required to select a preexisting reference within the model.

Figure 5 provides an example of a feature configuration using Ecore.fmp. B is an optional feature that is instantiated; C is a mandatory feature that is not instantiated, hence the red circle indicating an error; D is a [0..*] prototype that has one instance; E is a [1..*] prototype that has no instances, thus it has a red circle indicting an error; F is a [0..5] prototype with an instance; and G is a [2..5] prototype with less than 2 instances, therefore a red circle is shown indicating an error. At this point in the configuration, the optional feature B can be unchecked to remove the instance and features C-G can be cloned by double clicking on the their respective icons.

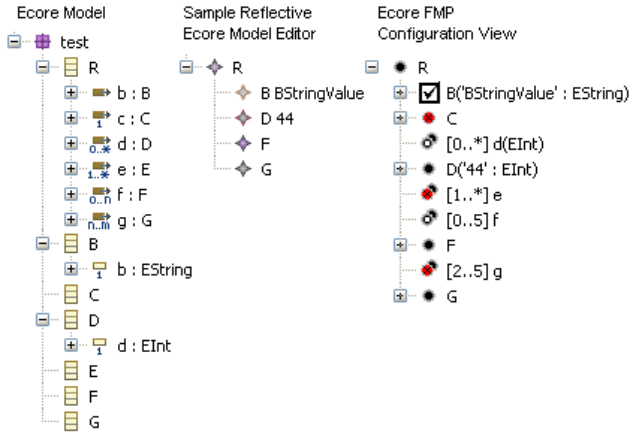


Fig. 5 Ecore.fmp Feature Configuration Example

6 Advanced Concepts

The following sections address some advanced issues that arise due to the semantic mismatch between class diagrams and feature models. Thorough investigation of the semantic mismatch remains future work.

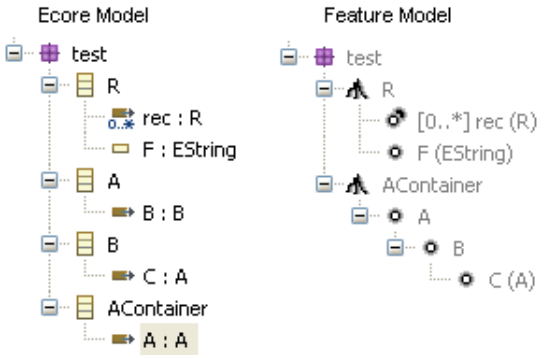


Fig. 6 Cyclic Containment Examples

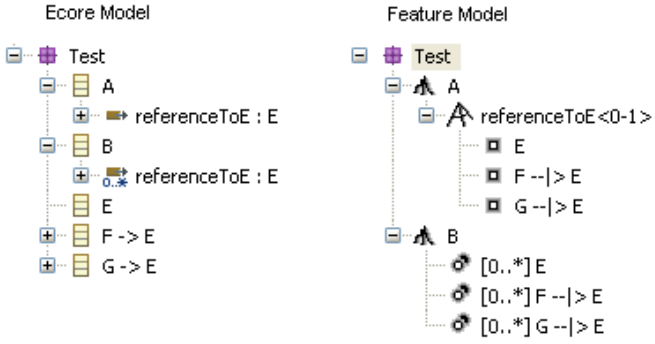


Fig. 7 Class Subtyping with a Containment Reference

6.1 Cyclic Containment

Cyclic containment refers to the situation where a feature contains, at any level in its hierarchy, itself through either direct or indirect containment. Figure 6 below shows both of these cases on the left in Ecore and the way they are interpreted as a feature models on the right. In the first case, R contains a containment reference to itself `rec`. In the second case the feature A, which is contained by the feature AContainer, contains the feature B that has the containment reference C back to A. In both cases the standard interpretation algorithm would execute in an infinite loop because of the cycle that forms when retrieving the children of the elements. To prevent infinite recursion, cyclic containment is detected by traversing the containment hierarchy upwards. Once detected, rather than adding a feature with representation `Class`, a feature with representation `Reference`, which does not contain any children, is added in the same way it is done for a non-containment reference thus ending the recursion at the first point it is detected. In the case of feature configuration, the upward branch traversal is stopped when an instance is reached. If none of the elements in the cyclic containment are a root feature, then the user is required to annotate the desired EClass that should be the root feature using the annotation `root` as discussed earlier.

6.2 Feature and Class Inheritance

The concept of class subtyping and feature modeling refers to features that represent a class with one or more sub types. In instances where a feature has a non-containment reference to such a class, the class and all its subclasses are shown as explicit references with the cardinality being inherited from the super class. Figure 7 shows both the Ecore representation and corresponding feature model of a class subtyping example with a containment reference. For containment references, there are two cases. One case, as shown as feature and Class A in the figure, is a derived feature group, which occurs when the reference has a defined upper bound. This will be explained below in more detail. The other case, element B in the figure, is a containment reference that has a * (unlimited) upper bound. This case is handled in the same way as non-containment reference due to the fact that it is no longer a true feature group, see below, and an infinite number can be replicated with no restrictions. Using the feature model to add a supertype transforms an attribute feature or reference feature into a Class Attribute or Class Reference type feature, respectively, before adding the supertype.

6.3 Feature Group - Derived and Annotated

As defined in [8], feature groups exist when a feature represents a choice among the elements within the group. In Figure 7, E, F, and G are the elements of the Feature Group belonging to A. This means A has a choice of selecting either no elements or one of E, F, or G.

Regarding Ecore.fmp, there are two type of feature groups that are interpreted from the Ecore model: derived feature groups and annotated feature groups. Derived feature groups are containment references with a finite upper bound that refer to a class that has one or more sub types. The reasoning behind this, as alluded to earlier, is that any time there is a reference to a class such as this, an implicit choice exists regarding what specific instance to select. This feature group is entitled derived because the Ecore.fmp determines the existence of this type of feature group by looking for the references described during the calculation of a Class feature's children.

An annotated feature group, on the other hand, are classes that are annotated with the annotation `featureGroup` or by using the feature model command *Add Feature Group*. Figure 8 provides an example of a class, R, annotated with `featureGroup`. The class R has a reference to the class A and three boolean attributes B, C, D. The annotation `featureGroup`, with `lowerbound` set to 1 and `upperbound` set to 3, specifies that a correct instance of R will have at least 1 and at most 3 structural features present (non-null or `true`). Using annoation `featureGroup` is necessary because Ecore does not provide an appropriate mechanism for expressing the group constraint. An alternative way of modeling a feature group constraint is by using a constraint language such as Object Constraint Language (OCL) [14].

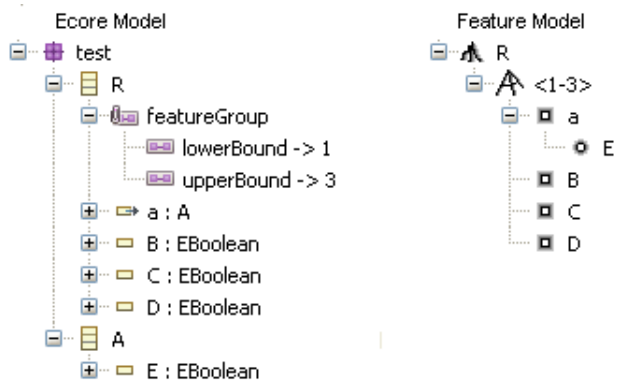


Fig. 8 Ecore Model Annotated with featureGroup

7 Prototype Implementation

Ecore.fmp is implemented as two Eclipse plug-ins and it uses Eclipse Modeling Framework (EMF). The first plug-in, `ca.uwaterloo.gsd.ecore.fm`, contains the Ecore.fm metamodel and its implementation, including feature modeling and configuration operations. This plug-in does not contain any graphical components and is meant for those who want to deal with the metamodel in isolation. The second plug-in, `ca.uwaterloo.gsd.ecore.fmp` adds the feature modeling and configuration views.

The feature modeling view works in conjunction with any Ecore class model (designated with the .ecore extension) opened using the standard Ecore editor. It displays the class model as a feature model and it supports feature model editing through feature addition, removal, and modification commands as well as similar commands for feature groups. These commands are operated on the feature model and are reflected instantly in the Ecore model. This view can be activated by right clicking on any element within an Ecore editor and selecting *Show Feature Modeling View* action.

The feature configuration view operates on any EObject and allows configuration of the instances associated with it and the children belonging to its hierarchy. Ecore provides a sample reflective object model editor that operates on .xmi object model files. The feature configuration view can be activated from this model editor by right clicking on any element within it and selecting *Show Configuration View* action. Once done, a feature configuration is shown that reflects the elements existing in the object model. Furthermore, commands can be performed in a similar fashion to the feature configuration component from [4], allowing for the addition, removal, and modification of instances. There is also very basic model validation present such that missing mandatory instances or insufficient quantity of instances are shown as red. For example, a reference of [1..1] or [1..*] with no instances will show red and the user will know that instances should be created in order to satisfy the cardinality constraints. An interesting side effect of this view is that because Ecore class models are comprised of EObjects, the feature configuration view can also be used to modify class models.

8 Related Work

The FeaturePlugin [4] was used as a template for the Ecore-.fmp . The Ecore.fmp uses many of the same notations and display elements as the FeaturePlugin. The key difference between that and the one described in this report is that the FeaturePlugin focuses strictly on feature modelling in an isolated context. That is, users are dealing with feature models and configurations as the only artifacts of interest. The Ecore.fmp Plug-in allows modeling and configuration but operates within the context of an Ecore model. As such, given the existence of an Ecore file or an Ecore instance file, the Ecore.fmp can accomplish the same tasks that the FeaturePlugin can. Future work is to allow creation of an Ecore file or Ecore instance file given a feature model or configuration without a preexisting model. Other tools have been released for feature modeling, such as CaptainFeature [6] and Pure::Variants [9], but none support cardinalities. Pure::Variants does support constraints and constraint-based configuration, something that is future work for the Ecore.fmp.

Asikainen et al. propose an ontology for feature models, called Forfamel, that attempts to acheive a unified conceptual foundation for feature modeling and configuration [5]. It synthesizes existing feature model definitions and tools and provides extensions to enable easy reuse. The feature modeling and configuration ontology used by the Ecore.fmp is based more on the traditional (non-forfamel) definitions discussed in [7,8]. In Ecore.fmp, entities in both a feature model and configuration are termed and treated as features; however, Forfamel distinguishes between the two and considers them “instances of different meta-entities”. In Ecore.-fmp, entities in feature models and configuration are distinguished by checking the value of the `ecoreInstance` reference: null indicates feature model, an object indicates configuration. Forfamel also has features/subfeatures unaware of their roles within the model, that is, features are unaware if they are a grouped, solitary, or root feature. Ecore.fmp requires that features have this knowledge in order to properly manage the bi-directional mapping and to render them correctly within their respective views. Lastly, Forfamel excludes unbounded (infinite) cardinalities. Infinite and unbounded cardinalities are used frequently in class modeling, so it was necessary for the Ecore.fmp to incoporate them.

9 Future Work

At this point, the feature modeling and feature configuration of the Ecore.fmp is dependent on an Ecore model. It would be more convenient if a modeler could create a feature model or a configuration of a model from scratch and then have the corresponding Ecore file created automatically. This should not be too onerous to implement because the mappings from

Feature model to Ecore are already defined. Following these rules, the Ecore file could be generated and the developer could then edit the Ecore model or continue editing the feature model.

A more involved addition to the plug-in is to support constraints for modeling and configuration. This will likely be accomplished in a similar way that Pure::Variants handles it, specifically through a Prolog-based constraint solver. It will need to enforce constraints between features and more complex interaction constraints.

Another area of work to be considered is the idea of model/feature interactions that can be facilitated via the plug-in. This entails using preexisting work on merging and interactions of feature models and/or Ecore models in order to perform a number of interesting tasks such as model comparisons and model merging.

Lastly, the semantic mismatch discussed in Section 6 is a very interesting problem and future work should be conducted to investigate this more thoroughly.

References

1. Ecore metamodel. http://help.eclipse.org/help32/topic/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/doc-files/EcoreRelations.gif
2. Eclipse modeling framework project (emf) (2008). http://www.eclipse.org/modeling/emf/
3. Omg's metaobject facility (2008). http://www.omg.org/mof/
4. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: feature modeling plug-in for eclipse. In: eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, pp. 67–72. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/1066129.1066143
5. Asikainen, T., Mnnist, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: 10th International Software Product Lines Conference (2006)
6. Bednasch, T., Endler, C., Lang, M.: Captainfeature (2004). https://sourceforge.net/projects/captainfeature/
7. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
8. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. In: Software Process Improvement and Practice, special issue of best papers from SPLC04, vol. 10, pp. 7 – 29 (2005)
9. pure-systems GmbH: Variant management with pure::consul. Technical White Paper (2003). http://web.pure-systems.com
10. IBM: Rational software modeler (2008). http://www-306.ibm.com/software/awdtools/modeler/swmodeler/
11. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University (1990)
12. Microsoft Corporation: Microsoft office visio 2007 product overview (2008). http://office.microsoft.com/en-us/visio/HA101656401033.aspx
13. No Magic Inc: MagicDraw (2008). http://www.magicdraw.com/
14. Object Management Group, Inc.: Object constraint language (2008). http://www.omg.org/technology/documents/formal/ocl.htm