Implementing an MDA Approach for Managing Variability in Product Line Construction Using the GMF and GME Frameworks

Hugo Arboleda[1,2], Rubby Casallas[1], Jean-Claude Royer[2]

[1] Software Construction Group, University of Los Andes, Carrera 1 N° 18A 10, Bogotá, Colombia
[2] OBASCO Group (INRIA & LINA), Ecole des Mines de Nantes, La Chantrerie, 4 rue Alfred Kastler, 44307 Nantes, France
{hugo.arboleda, jean-claude.royer}@emn.fr, rcasalla@uniandes.edu.co

Abstract. In this paper, we present a comparison of two implementations of our proposed MDA approach for managing variability in a software product line. The implementations correspond to two representative frameworks based on the Model Driven Engineering (MDE) principles. These frameworks are the Graphical Modeling Framework (GMF) and the Generic Model Environment (GME). We built the core assets of the product line and we generated applications using the two different frameworks. The core assets that we built are: feature models, metamodels, *mapping* models, and three different types of transformation rules. We built the transformation rules using two different languages: the ATLAS Transformation Language (ATL) in the context of GMF and, the Embedded Constraint Language (ECL) in GME.

Keywords: Model Driven Architecture, Variability, Software Product Lines, and Model Transformation.

1 Introduction

A software product line (SPL) is a set of software systems that satisfies specific needs for a segment of the market. In a SPLs development approach, we can create new products of the line reusing components called core assets. While the creation of core assets is part of the domain-engineering process, the creation of applications reusing the core assets, is part of the application engineering process [1]. The management of variability in an SPL is related to the necessary activities and assets: (1) to express the common and variable characteristics of a SPL, and (2) to build applications that include common characteristics, and a subset of the possible variable characteristics. Currently, feature models [2] are a standard *de facto* used as a mechanism to support variability.

Model Driven Architecture (MDA) [3] is an approach of generative reusability. The main intention of MDA is not the creation of product families; nevertheless, the separation of domains and its generative nature make MDA a useful approach for the creation of SPLs. Results of recent researches show how the use of MDA in

conjunction with SPL engineering (MD-SPL) facilitates the definition of SPL creation processes [4], [5], [6].

We presented a scheme in [7] to manage the variability in SPL construction processes using an MD-SPL approach. In the same paper, we presented a related work section, which we have omitted here to respect the length restrictions. In our scheme, we separated concepts related to product lines in different domains: (1) the business logic domain, (2) the architecture domain, and (3) the platform technological domain. We created metamodels, feature models, *mapping* models, and transformation rules as core assets for each domain. This scheme enables us to transform one initial source model into different (variable) target models for obtaining variable SPL members. In order to guide the generation of variable target models, we created three types of transformation rules: (1) base rules, (2) control rules, and (3) specific rules. However, these core assets are not enough to generate complete applications. Since some functional requirements are not generated during the automatic transformation processes, we manually complemented the applications in the application engineering process.

In this paper, we present the implementation of this approach using two representative frameworks based on the Model Driven Engineering (MDE) principles. We present a comparison of the features provided by the frameworks used to perform the implementation. These frameworks are the Graphical Modeling Framework (GMF) [8] that implements the OMG-MDA standard, and the Generic Model Environment (GME) [9] that implements the Model Integrated Computing (MIC) standard [10]. For building transformation rules, we use the ATLAS Transformation Language (ATL) [11] for the GMF implementation, and the Embedded Constraint Language (ECL) [12], that supports the concept of aspect model weaving combined with the idea of model driven transformation, for the GME implementation. We base our experimentation on the work of Bézivin et al. [13] for defining an interoperability scheme to exchange models between the two frameworks. However, we compare features like meta-metamodel (M3) facilities, environments for creating metamodels and generating model editor plug-ins, support for (meta)models composition, and transformation languages. This work is part of the AMPLE project [14]. The aim of AMPLE is to provide a SPL development methodology that offers improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their traceability during SPL evolution.

The paper is organized as follows. Section 2 presents a brief introduction to the frameworks and the transformation tools that we use for the two different implementations. Section 3 introduces the application context that allows us to illustrate the two implementations. Section 4 gives a summary of the approach for managing variability in MD-SPLs construction processes and the two different implementations describing the process followed to create the core assets. Section 5 describes the comparison of the features provided by the frameworks for this implementation. Finally, section 6 draws a conclusion and some future work.

2 MDE Frameworks and Transformation Languages

In this section, we introduce GMF, GME, and the two model-to-model transformation languages we use: ATL in the context of GMF and ECL in the context of GME.

2.1 The Graphical Modeling Framework and the ATLAS Transformation Language

GMF provides a generative component for developing graphical editors using the Eclipse Modeling Framework (EMF) [15] and the Graphical Editing Framework (GEF) [16]. EMF is a modeling framework for building tools and other applications based on data models. EMF provides tools to produce a set of Java classes for the model that makes it possible to visualize and edit them. GEF allows developers to create graphical editors from an existing application model. Thus, with EMF (using the Ecore metamodel) and GEF, GMF proposes and supports a process for building Eclipse-based functionalities. The concept of a graphical definition model is the core of GMF. This model contains information related to the graphical elements that will appear in a GEF-based runtime environment.

Currently ATL is a standard Eclipse solution to define transformations, and it is one of the major components of the Model-to-Model Eclipse project. ATL is a hybrid of a declarative and an imperative language and is used to build transformations composed of rules that define how elements of a source model are transformed into elements of a target model. ATL supports the creation of two kinds of rules: (1) matched rules (declarative programming), and called rules (imperative programming). The matched rules constitute the core of ATL declarative transformations. The called rules provide imperative programming facilities; they have to be explicitly called to be executed.

2.2 The Generic Model Environment and the Embedded Constraint Language

GME is based on the MultiGraph Architecture (MGA), which is a part of MIC. GME is a toolkit for creating domain-specific modeling and program synthesis environments, which use MetaGME as their metamodel (M3) for building domain models. Configuration of domain models is accomplished through metamodels (M2) specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain. This defines the family of models that can be created using the resultant modeling environment. GME supports multiple *aspect* modeling. It allows metamodel composition for reusing and combining existing modeling languages and language concepts.

ECL is a procedural transformation language. ECL is implemented into a model transformation engine called the Constraint-Specification Aspect Weaver (C-SAW) [12]. ECL is an extension of the OMG’s Object Constraint Language (OCL), it extends OCL by providing a series of operators for changing the structure or constraints of a model. ECL provides features such as collection, model navigation and a set of operators to support model aggregations, connections, and transformations. The structure of ECL modules includes strategies and aspects. A strategy defines a specific transformation procedure, and an aspect is a special strategy that serves as the entry point of a transformation. An ECL specification consists of many strategies, and a strategy can call other strategies. A strategy is applied by traversing a model and matching elements of the model that satisfy a predicate.

3 Application Context

As case study, we created an SPL for the Cupi2 project [17]. Cupi2 is a project of the software construction group of the University of Los Andes from Colombia that proposes a new approach to teach/learn computer programming. The approach is problem based: the problems used as learning exercises are classified by levels. There are 18 levels and each one adds new concepts to teach. As an example scenario, we use level 7 that includes, among other topics, the algorithms to perform searches and orderings using collections. All the Cupi2 examples are stand-alone applications without complex non-functional requirements; they are developed using the same technological platform, in this case Java. All the examples are structured by three components: the kernel, the user interface, and the tests. The kernel component implements the concepts of the business logic. The user interface component implements visualization of the information and the interaction between the user and the kernel component.

The kernel concepts and their relationships can be functionally represented and manipulated by data structures of type Group. A Group always has a main element that groups the other elements of the kernel. For example, in a Music Store application, it should be a MusicStore Group that assembles Discs, and each Disc assembles Songs. All the kernel elements have a set of properties and these properties are possibly related to other elements of the kernel. Finally, each kernel element is responsible to make its information persistent.

The graphical user interfaces (GUI) use panels, lists, labels, images, and radio buttons, among others. All the GUI elements are grouped in views of different types. There are two types of views that are mandatory for any Cupi2 application: (1) the MainView, and (2) the ExtensionView. The MainView is in charge of communication between the kernel component and the other views and groups all the other views of the GUI. The ExtensionView contains buttons that the student can use to add new functionality as part of the exercise.

At the same time that we identify commonalities, we identify variability as well. The variability in the Cupi2 examples is related to the algorithms that manage the groups in the kernel component, and the presentation of the GUI. In the kernel component there are variable elements related to data structures, algorithms, and services to persist the different assemblies. The data structures that represent the groups can be fixed size or variable size containers, or can be linear structures like simple lists or doubly connected lists. Each type of data structure can be manipulated using different algorithms; for example, it is possible to manipulate a data structure with algorithms to make insertion, search, and ordering, among other services. Different implementations can be used to make information of the kernel elements persistent; for example by managing flat files, structured files or using object serialization. There is no a unique way to represent the kernel elements in the GUI. The user can select one or several types of views to represent the kernel elements. The types of view are: main, extension, set, search, information, and aggregation view.

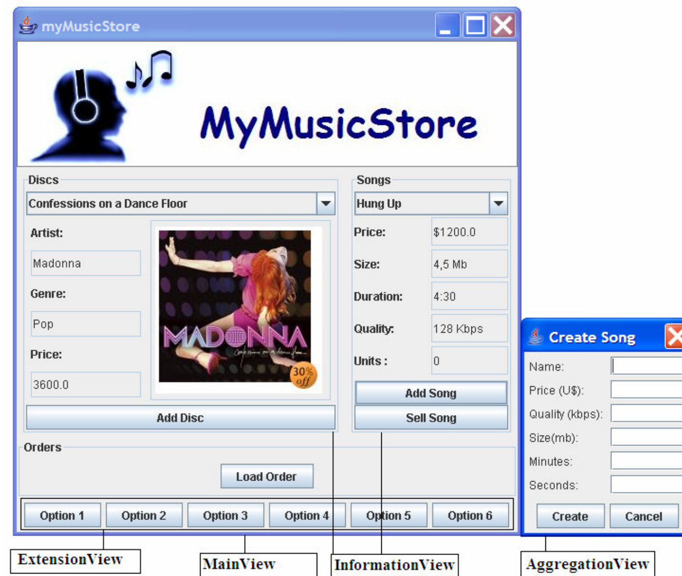


Fig. 1. A possible graphical user interface for a Music Store application

The MainView assemblies other views and it is responsible for the communication between the kernel component and the user interface component. The SetView is in charge of presenting a set of grouping elements using components such as lists or tables. The SearchView is used to enter parameters to search in groupings. The InformationView is used to show the particular information associated to the attributes of a kernel element of the problem, including images. The AggregationView is used to enter particular information associated to the attributes of new kernel elements. Fig. 1 presents an example of a possible GUI for the Music Store example showing four types of views.

4 The Variability Management Approach

In this section, we present a summary of the approach for managing variability in MD-SPLs construction processes and we describe how the approach was implemented using GMF/ATL and GME/ECL. We separate concepts related to a SPL in domains to manage variability. These domains are the business logic (high abstraction level), the architecture domain, and the technological platform domain (low abstraction level). Our strategy follows the MDA approach and it is based on the automatic transformation (refinements) of models until obtaining executable applications. We start from a business logic model, we transform it into an architectural model, after that we refine it into a model in the technological platform domain and finally we generate source code from it. To achieve this, we build as core assets (1) metamodels for each domain, (2) feature models for each target domain, (3) mapping models, and (4) three types of transformation rules. Fig. 2 presents the domain-engineering process.

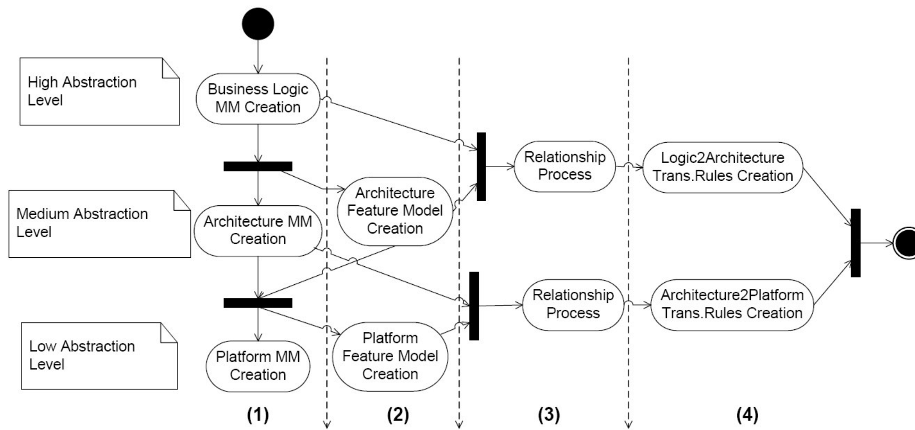
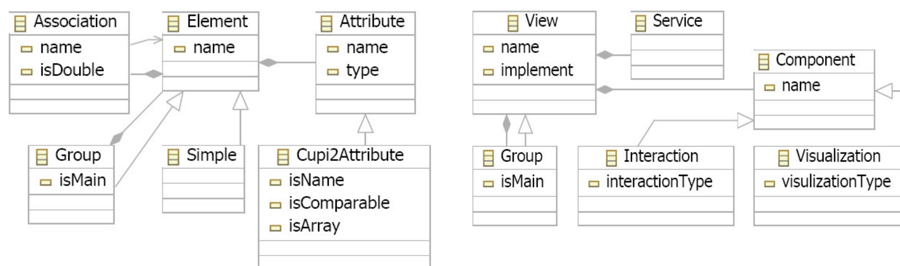


Fig. 2. Domain-Engineering process.

4.1 Metamodels Definition

For the creation of the Cupi2 (Level 7) SPL we defined three different metamodels as core assets: business logic, architecture, and technological platform metamodel. Business logic metamodel includes the essential concepts of the problem; architecture metamodel includes concepts of the architecture of the applications including the user interface; finally, the technological platform metamodel includes concepts of the language as for example packages, classes, methods and attributes. For the last one, we created a simplified Java metamodel.



(a) Business logic metamodel (b) Business logic architecture metamodel

Fig. 3. Business logic and GUI architecture metamodels

The business logic metamodel has concepts to describe a set of elements related between them using Group structures. Each element can have a set of Attributes (see Fig. 3a). The architecture metamodel includes design concepts of the Cupi2 examples. The business logic architecture metamodel introduces the concept of Service. Services are needed to manipulate the data structures of the examples. The GUI architecture metamodel includes the concepts of graphical and interaction elements that will be part of the GUI of generated applications. Fig. 3 presents parts of the created metamodels. For the metamodel definition activity, we focus on the

meta-metamodel (M3) concept facilities, the environments for creating metamodels, and the generated model editor plug-ins.

4.1.1 Metamodel Definition in GMF

For the metamodel definition, GMF is based on the Eclipse Modeling Framework (EMF), which is a Java implementation of a core subset of OMG-MOF called Ecore. Using GMF, we created EMF models using the Eclipse Ecore diagram editor, which allows us to create models similar to UML class diagrams. The structural organization of Ecore (M3) includes meta-concepts like `EPackage`, `EClass`, `EAttribute`, `EReference`, `EOperation`, `EDataType`, `EEnum`, and `EEnumLiteral` among others. The Ecore metamodel admits multiple inheritance, therefore one meta-class can have different super meta-classes; and the `EReference` concept is used to define association or aggregation relationships between classes. Using these concepts, we built one metamodel for each domain. We created the plug-in editors to edit models conform to the EMF metamodels using the GMF framework for building Eclipse-based functionalities. This framework separates the definition of the domain model (M2), the graphical model associated (concrete syntax), and the tools to manipulate domain concepts with the associated concrete syntax. Thus, the domain model is not coupled with concrete syntax, and different concrete syntax can be associated. Finally, the generated Eclipse plug-in editors are Java plug-ins; thus, we can modify these to include, for example, constraints associated to the domain model.

4.1.2 Metamodel Definition in GME

MetaGME is the meta-metamodel (M3) for the GME framework. MetaGME is a UML profile with meta-concepts such as `Project`, `Folder`, `Model`, `Atom`, `Attribute`, `Connection`, `Proxy`, `Reference`, and `Aspect` among others. A `Project` contains `Folders`; a `Folder` contains `Models`. A `Model` is composed of `First Class Objects` (FCO). `Relationships` between `FCO` are `Connections` or `References`. A `Connection` can only express relationships between objects contained by the same `Model`. `References` and `Proxies` are useful to express a relationship between objects contained by different models (but contained by the same `Project`). The `Proxies` are references pointing to concepts defined elsewhere in a metamodel. Finally, `Aspects` provide visibility control; they define points of view on models. Using the MetaGME concepts, we built one metamodel for each domain. However, we reused concepts of different metamodels using `proxies`. Thus, we created `FCOs` (`Models`, `Atoms`, and `Connections`) in the business logic metamodel, and we reused them in the architectural metamodel. For example, we reused the Element concept (`Model`) and the name concept (`Attribute`) defined in the business logic metamodel, in the logic architecture metamodel (Fig. 3b). Thus, we created metamodels using (composing) concepts of other metamodels, and adding information to refine the concepts when it was needed. We used metamodels to generate the domain-specific environment (model editors). We created one `Aspect` for each metamodel and related one icon to each `FCO`, thus each icon represents the concrete syntax associated to each metamodel concept. Finally, the generated

domain-specific environments are represented by XMI code that is interpreted by the GME framework.

4.1.3 Metamodel Definition Comparison

In [13], there is a proposal of an interoperability scheme to exchange models that conform to the Ecore metamodel and the MetaGME metamodel. However, there is not a one-to-one relationship between the concepts that each environment provides.

Both meta-metamodels offer concepts to organize elements, *i.e.*, `EPackage` in Ecore or `Folder` in GME. They also offer concepts to represent domain concepts, `EClass` and `EAttribute` in Ecore, and `Model`, `Atom` and `Attributes` in MetaGME. `EClass` is the only concept in Ecore to define domain concepts, while in MetaGME `Models` are complemented with `Atoms`. The `Atoms` are specialized `Models` which can not group others `Models` or `Atoms`. Thus, `Models` aggregate concepts, and `Atoms` are always "aggregated" concepts. One `EAttribute` is related just with one `EAttribute` in Ecore, but in MetaGME one `Attribute` can be reused to be related with different `Models` or `Atoms`. Regarding the way of associating domain elements, in Ecore the `EReference` concept is used to define association or aggregation relationships between `EClasses`. In MetaGME `Relationships` between domain concepts can be `Connections` or `References`. Using `Connections` is possible to relate objects included in the same `Model`. Using `References` is possible to express a relationship between objects contained by different models. The Ecore metamodel does not define equivalent concepts for MetaGME `Proxies`, `Aspects`, and `Constraint` concepts, and does not allow the association of graphical information with the metamodels. The MetaGME metamodel does not define equivalent concepts for the Ecore `EProperties` concept.

For us the Ecore metamodel looks essentially as an UML class diagram, and the MetaGME metamodel has extensions for composing metamodels using `Proxies`, `Aspects`, different types of relationships and inheritance, admitting the reuse of `Attributes` and the creation of constraints directly associated with metamodel concepts. We found that the main problem with metamodel composition using GME is that metamodels can be *low cohesive; by* reusing concepts of different domains, or domains concepts of different abstraction level.

Regarding the environments to create metamodels, both environments provide similar graphical facilities. The MetaGME offers the option to associate graphical information to domain concepts. GMF complements the EMF facilities by also offering facilities to define graphical information. Finally, once the metamodels are created, by using both frameworks it is possible to generate graphical environments to create models that conform to the defined metamodels. We found the GMF plug-in process creation more complex than the GME editing model creation. However, the model editor plug-ins generated by GMF are easily modified/extended since they are built using Java source code; while the GME environments for editing models are defined using XML code that is interpreted by the GME framework, which is more complex to modify/extend.

4.2 Feature Models Definition

We created feature models based on the characteristics of the applications in the architecture and the technological platform domains. To define our feature models, we used concepts such as feature nodes, feature groups, and cardinality. Fig. 4a presents a part of the feature model for the architecture domain. These features are related to the commonalities and variability identified in section 2 of this paper. Fig. 4b presents feature nodes of the technological platform domain (Java) feature model. The relationship between the features nodes implies that nodes selected in the architectural feature model constrained the selection the user can do in the platform feature model. For example, selection of PrintWriter or BufferReader features (Fig. 4b) can be done only if the Files feature (Fig. 4a) is selected before.

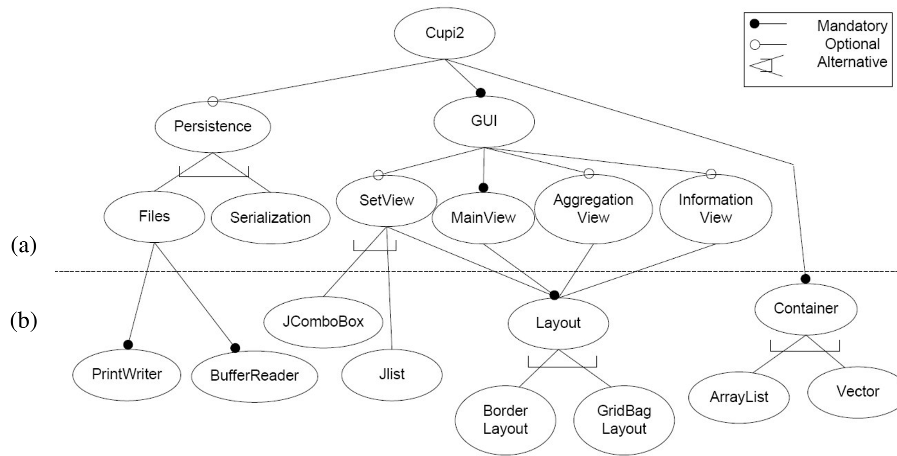


Fig. 4. Feature model for Cupi2 domains

4.3 *Mapping* Models and Transformations Rules Definition

Users define their preferences by means of the feature models; they have to make selections on features before the initialization of the transformation processes. The selection is done by associating (source) model elements and (target) feature nodes. Thus, for example for the music store application, the classes Disc and Song, both conform to the Element concept (Fig. 3a); one user may want to design the persistence for Disc elements using files, and the persistence for Song elements using serialization. For this, the user must relate the Disc element with the Files feature node (Fig. 4), and the Song element with the Serialization feature node.

Transformation rules are defined in terms of metamodel concepts. It implies that a same source model is always transformed into the same target model. Since we need to transform the same metamodel concept into different target concepts according to the features related, we need to create different rules. For example, for the same music store application, even when the Disc and Song elements conform to the Element concept, different transformation rules must be created to transform Elements to obtain the Files feature or the Serialization feature.

For doing this, we created three types of transformation rules. These types of rules are: (1) base rules, (2) control rules, and (3) specific rules. In order to guide the creation and posterior execution of these transformation rules, we created new composed models containing relationships between metamodel concepts and feature nodes, and new composed models containing relationships between model elements and feature nodes. We call the first *meta-mapping* models, and the second *mapping models.*

The meta-mapping models link metamodel concepts to feature nodes generating a new composed model. Each link means that one source metamodel concept can be transformed to obtain the target feature node. Thus, these meta-mapping models are core assets used as design elements for constructing the different transformation rules. Using the meta-mapping models, we can identify (1) the metamodel concepts that have to be always transformed into the same target feature, and, (2) the metamodel concepts that can be transformed into different (variable) target features. For the first case, we create base rules; and for the second, we create control and specific rules. Thus, the base rules allow us for the generation of the commonalities of the product line, and the control and specific rules the variability. Fig. 5 presents a meta-mapping model between the business logic metamodel and the feature model of the architecture domain. In Fig. 5, the link between Group and the mandatory feature MainView indicates that a MainView is always created for Group elements (isMain == True). For this connection, we create a base rule. The links between Element, and the Serialization and Files feature nodes indicate that an element of type Element can implement its persistence using Files or Serialization methods. For these connections we created one control rule and two specific rules.

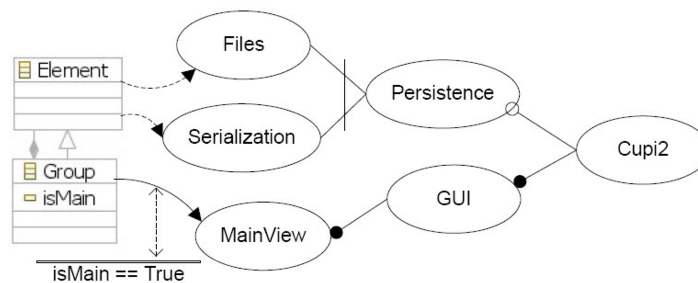


Fig. 5. Meta-Mapping model between the business logic metamodel and the architecture features model

We link model elements with feature nodes for creating new composed mapping models in the application engineering process. These models are created when the user selects model elements and relates them to selected features to guide the generation process. Since these mapping models are inputs to the transformation processes, it is mandatory to use a mapping metamodel as core asset in the domain-engineering process. We use the ATLAS Model Weaver (AMW) [18] metamodel for the GMF implementation, and we create one mapping metamodel for the GME implementation.

For this activity we focus on the (meta)model composition facilities provided by the two frameworks needed to create mapping models, and the transformation language facilities for creating our three types of transformation rules.

4.3.1 Mapping Models and Transformation Rules Definition in GMF

For the GMF implementation we create the (meta)mapping models using the AMW metamodel and its plug-in. The AMW is integrated into Eclipse for establishing relationships (*i.e.*, links) between models. The created links are stored in a new model, called weaving model that conforms to a weaving metamodel. We create the transformation rules using the declarative and imperative facilities provided by ATL. We implement the base rules using declarative programming, and the specific rules using imperative programming. The control rules are implemented in a mixed way, it means, they have a declarative section and an imperative section. The ATL modules we created have as input the source model that will be transformed, and a mapping model. The mapping model is used, in the control rules, to decide which transformation rule must be executed according to the feature linked to the element that is being transformed. Listing 1 presents an example of control and specific rules. The `setView` is a control rule. The declarative section is in the lines 2 to 4 and the imperative section is in the line 5. The `setView` rule searches in the mapping model the elements that have the feature `SetView` associated (line 2). For those elements a `View` is created (line 4) and the `addComponent` imperative rule is called (line 5). The `addComponent` rule creates concepts associated to the created `View` giving the characteristics needed for a `SetView`. The `addComponent` rule is a specific rule which has the imperative sentences in the line 9. When this rule is called from the control rule, a `Visualization Component` is created (line 8), the parameter type is assigned to this component, in this case `List` (line 8), and the created component is added to the previous created `View` (line 9).

```
(1)      rule setView{
(2)        from link: mapping!Link(link.feature=='SetView')
(3)          using{ element: kernel!Element = link.getElement(); }
(4)        to   target: Architecture!View()
(5)          do{ addComponent(target, #list);       }
(6)      }
(7)      rule addComponent(view: View, t: Type){
(8)        to component: Architecture!Visualization( type<-t )
(9)        do{ view.attribute.add(component); }
(10)     }
```

Listing 1. ATL control and specific transformation rules

4.3.2 *Mapping* Models and Transformation Rules Definition in GME

We create meta-mapping models without tool support because using GME it is not possible to link metamodels concepts (M2) and model elements (M1). To have a mapping metamodel, we use the `Reference` and `Proxy` MetaGME concepts for composing models. `References` are similar to the pointer concept found in various programming languages. Thus, the metamodel includes `Proxies` of all the concepts that can be linked with Feature Nodes, and a common *"super-type"* `Model` concept (MetamodelElement in Fig. 6) for all of them. This concept has a `Connection` (aggregation) with one `Reference` concept that refers the FeatureNode

concept of the feature metamodel. Thus, the model elements can be related through `References` with different FeatureNodes in the application engineering process. Fig. 6 presents the GME mapping metamodel.

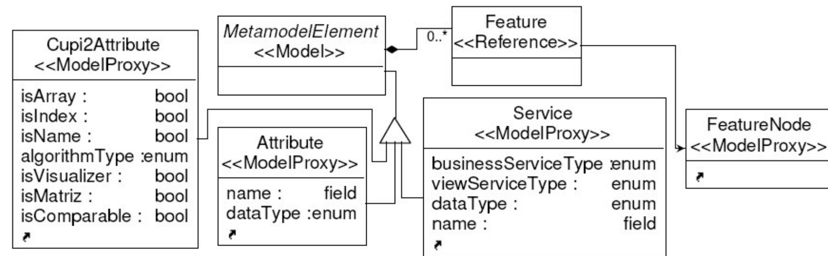


Fig. 6. GME mapping metamodel

Once we created the meta-mapping models, we built transformation rules using ECL. Transformation rules in ECL are written as `Aspects` and `Strategies`. Thus, we create that we called base strategies to create the commonalities, specific strategies to create variability characteristics, and control strategies to decide which specific strategy is used to create variable functionality.

```
(1)      strategy viewArch(name : String){
(2)        viewArchM := rootFolder().addModel ("ViewArch", name);
(3)        mainView := viewArchM.addModel("Group","MainView");
(4)        elems := this.models()->select (m|m.kindOf()=="Group" or
             m.kindOf()=="Simple");
(5)        elems->transElem(mainView);
(6)      }
(7)      strategy transElem(mainView: model){
(8)        featureNodes := self.modelRefs("Features");
(9)        featureNodes->select (m|m.name()=="SetView")-> transElemSetView(this,
             mainView);
(10)     }
(11)     strategy transElemSetView(elem: model; mainView: model){
(12)       setView := mainView.addModel("Group", elem.name());
(13)       setView.AddViewComp(setView,"List", "List", "Interaction");
(14)       elem.models("Cupi2Attribute")->select(m|m.getAttribute("isComparable"))
             ->AddButtToManageList(setView);
(15)     }
```

Listing 2. ECL base, control, and specific strategies

Listing 2 presents an example of base, control and specific strategies. The `viewArch` strategy is a base strategy to transform business logic models to GUI architecture models. In line 2 an element that conforms to `ViewArch` is created. This is the root of any `ViewArch` model. As part of the transformation logic, the `mainView` is always created (line 3). Finally a control strategy is called (line 5) to transform elements that match a condition defined in the line 4. The `transElem` control strategy (line 7) explores business logic models searching `References` from `self`, which is a `Group` or a `Simple` concept, to different feature nodes. For this example the `References` to a `SetView` feature node are searched (line 9). For each reference found the `transElemSetView` strategy is called (line 9). Finally, the `transElemSetView` specific strategy (line 11) transforms items that conform to the `Element` concept to

`SetView` features. This means that the rule creates target elements that allow having a GUI with a `SetView` (lines 12-14).

4.3.3 *Mapping* Models and Transformation Rules Comparison

The meta-mapping models and mapping models creation are model composition activities. In our approach, model composition appears in three contexts: (1) composition of metamodels, (2) composition of metamodels with models, and (3) composition of models. The two first are done in the domain engineering process. The first takes place in the metamodel definition activity (section 4.1) and the second, in the meta-mapping models creation. The third takes place in the mapping models creation in the application engineering process.

The three composition activities are not supported by the Ecore metamodel in the contexts of GMF. However, it is possible to do the same using AMW. We found that using AMW allows for composing models using a graphical interface to relate concepts of the source models, maintaining *cohesive* each one, and creating new weaving models with new domain specific targets.

In the context of GME, the MetaGME metamodel supports the composition of metamodels; however, the composition of metamodels with models is not supported, and the composition of models is supported using concepts provided by MetaGME to create a composition metamodel and thus create composed models. This is the case of the mapping metamodel that we created. It is important to remark that models can be composed just with other models in the same `Project`.

In the construction of transformation rules, we found ECL an intuitive language to define transformation rules. Furthermore, ECL supports the concept of aspect model weaving. Note that composition of models using ECL implies that the source models and the target model conform to the same metamodel. We found that even while the ECL language has sufficient number of operators for transforming/composing models, it does not support definition of declarative strategies, therefore implying that we always need to define the logic to traverse across the models. Thus, strategies for transforming models with ECL imply writing programs that navigate manually through the models that will be transformed or composed. Another thing is that ECL has some problems manipulating `boolean` type and control structures. The `boolean` type does not have symbolic representation for `boolean` elements defined in GME models, and `boolean` elements can not be defined in ECL programs. Furthermore, the control structures do not admit using else-if conditions, making the evaluation of grouped conditions less efficient.

Different to ECL, ATL facilitates the creation of imperative and declarative rules. Furthermore using ATL it is also possible to build rules that trigger the execution of specific rules. This is the case of the control rules. These can be used, as in ECL, to build composed models; besides, using ATL source models and the composed target models can conform to different metamodels. Finally, different to ECL at the moment, there are various ATL discussion groups, examples and available resources that support work with ATL.

4.4 Models-to-Text Transformations Rules Creation

We use Acceleo [19] as model to text transformation language in the GMF implementation. This tool is specialized in the generation of text files (code, XML, documentation) starting from models. Using Acceleo we take the EMF models and use templates for the source code generation.

For the GME implementation we have to create our own Java generator to navigate the (XML) GME models and generate Java code.

The Model-to-Text (Java) transformation facility is less supported in GME. Even when there are available technologies to access GME data, we found it restricted and poorly supported. GME components can be built using the Unified Data Model module (UDM) [20]. However these support tools provide restricted languages to manipulate models that do not allow the use of control mechanisms as loops and conditional statements, and the traceability management between models and the generated text.

Different to this, there are various versions of tools that support the model-to-text transformation construction process in the context of GMF. Examples of these are Acceleo and MOFScript [21]. These tools allow for generation of implementation code or documentation from models, generation of text from MOF-based models, manipulate control mechanisms as loops and conditional statements, and managing traceability between models and generated text. One important thing is the support for these tools. For example, currently the MOFScript language is a candidate in the OMG-RFP process on MOF Model-to-Text Transformation.

5. Conclusions and Future Work

In this paper we presented the implementation of an MDA approach for managing variability in product line construction using the GMF and the GME frameworks; and the comparison of the features provided by the frameworks for these implementations. The approach allows managing variability at *model level* for MD-SPLs construction. This means that we enlarge the SPL scope by making possible to generate different (variable) SPL members starting from the same initial business logic (platform independent) model, guided by a specific features selection. The implementation of this approach using two of the most representative frameworks and the followed comparison allows us to know each framework and take decisions about which of them is better for specific experimentations in the MD-SPLs field.

Currently we continue working on the presented approach focusing in (1) the constraint management for delimiting the SPL scope, (2) the inclusion of information of functional requirements in the metamodels, and (3) the traceability management required for maintaining core assets and generated applications. For this, we have chosen GMF for future experimentation. We found that GMF allow us to create cohesive metamodels, compose and manipulate these using the AMW, optimize the transformations using a declarative language, create transformation rules for transforming the same models using variable transformation logic, and transform models to source code supported by tools with necessary characteristics. In addition we found the current eclipse community support to be a valuable addition.

References

1. Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice, Vol. 10, No. 2. (2005) 143-169.
3. Kleppe, A., Warmer, J., Bast, W. MDA Explained: The Model Driven Architecture – Practice and Promise. Reading, Addison-Wesley (2003)
4. Estublier, J., Vega, G.: Reuse and Variability in Large Software Applications. In Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal (September 2005) 316-325
5. Santos, A.L., Koskimies, K., A. Lopes, A.: A Model-Driven Approach to Variability Management in Product-Line Engineering. In Nordic Journal of Computing, Volume 13, Issue 3 (September 2006) 196-213
6. Wagelaar, D.: Context-Driven Model Refinement. In Proceedings of the Model Driven Architecture: Foundations and Applications Workshop, Linköping, Sweden (June 2004) 189-203
7. Garces, K., Parra, C., Arboleda, H., Yie, A., Casallas, R.: Administración de Variabilidad en una Línea de Producto Basada en Modelos. In Proceedings of the Congreso Colombiano de Computación, Bogotá, Colombia (April 2007) CD-ROM. On line: http://educon.javeriana.edu.co/2ccc/
8. GMF, Graphical Modeling Framework site. On line: http://www.eclipse.org/gmf/
9. Davis, J.: GME: The Generic Modeling Environment. In Proceedings of the 18th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, USA (October 2003) 26-30
10. Ledeczi A., Balogh G., Molnar Z., Volgyesi P., Maroti M.: Model Integrated Computing in the Large. In Proceedings of the IEEE Aerospace Conference, Big Sky, USA (March 2005) CD-ROM
11. Jouault, F., Kurtev, I.: Transforming Models with ATL. In Proceedings of the Model Transformations in Practice Workshop, Montego Bay, Jamaica. (October 2005) 128-138
12. Gray, F., Bapty, T., Neema, S., Schmidt, D., Gokhale, A., Natarajan, B.: An approach for supporting aspect-oriented domain modeling. In Proceedings of the 2nd international conference on Generative programming and component engineering, Erfurt, Germany (September 2003) 151-168
13. Bézivin, J. Brunette, C. Chevrel, R. and Jouault, F., Kurtev, I.: Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). In Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development, San Diego, USA (October 2005)
14. AMPLE Project site. On line: http://ample.holos.pt/pageview.aspx?pageid=1&langid=1
15. EMF, Eclipse Modeling Framework site. On line: http://www.eclipse.org/emf/
16. GEF, Graphical Editing Framework. On line: http://www.eclipse.org/gef/
17. Cupi2 Project site. On line: http://cupi2.uniandes.edu.co.
18. Didonet Del Fabro, M., Jouault, F.: Model Transformation and Weaving in the AMMA Platform. In Proceedings of the Generative and Transformational Techniques in Software Engineering Workshop, Braga, Portugal (July 2005) 71-77.
19. Acceleo site. On line: http://www.acceleo.org
20. Magyari, E., Bakay, A., Lang, A., Paka, T., Vizhanyo,A., Agrawal, A., Karsai, G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling, Anahiem, USA (October 2003)
21. MOFScript site. On line: http://www.eclipse.org/gmt/mofscript/