Administración de Variabilidad en una línea de producto de software basada en modelos

Variability Management in a Model-Driven Software Product Line

Kelly Garcés[1,2], M.Sc., Carlos Parra[1,3], M.Sc., Hugo Arboleda[1,2], M.Sc., Andrés Yie[1,5], M.Sc., Rubby Casallas[1] Ph.D.

[1]University of Los Andes, Colombia; [2]Ecole des Mines de Nantes, France
[3] University of Science and Technology, France; [4]Vrije Universiteit Brussel, Belgium

Recibido para revisión 26 de Marzo de 2007, aceptado 15 de Junio de 2007, versión final 9 de julio de 2007

***Abstract*—La administración de variabilidad en una línea de producto tiene dos retos fundamentales: (1) la expresión de las características comunes y variables de la línea, y (2) la construcción de aplicaciones que incluyan las características comunes, y un subconjunto de las características variables. En este artículo presentamos una propuesta para administrar la variabilidad durante el proceso de construcción de Software Product Lines (SPLs) usando un enfoque de construcción de líneas de producto basado en modelos (MD-SPL). Para esto separamos conceptos relacionados con SPLs en diferentes dominios y construimos como activos de la línea modelos de rasgos, metamodelos y tres tipos de reglas de transformación para transformar modelos de un dominio origen a diferentes (variables) modelos en un dominio destino. Las reglas nos permiten generar incrementalmente las aplicaciones de acuerdo con una selección de rasgos realizada para cada dominio destino. Así, logramos ampliar el alcance de las SPLs, separar los dominios de manera que se disminuya la complejidad de crear aplicaciones con características variables, y generar aplicaciones automáticamente usando reglas de transformación. Para ilustrar la solución construimos una MD-SPL donde los productos corresponden a ejercicios pedagógicos para la enseñanza de programación de computadores.**

***Key Words*—Model Driven Architecture, Variabilidad, Lineas de producto de software, y Transformaciones de modelos**

***Abstract*—Variability management in Software Product Lines (SPLs) has two fundamental challenges: (1) the expression of common and variable features, and (2) the development of applications employing properly such features. In this paper, we present a Software Product Line based on Models (MD-SPL). We separate the concepts related to SPLs in different domains and we build core assets like feature models, metamodels, and three different types of transformation rules to transform models from a source domain to different (variable) models into a target domain.**

By using transformation rules, we are able to generate applications in an incremental process, guided by a set of features selected for each target domain. Thus, we manage to extend the SPLs scope, separate the domains diminishing the complexity to create applications with variable characteristics, and automatically generate applications using transformation rules. In order to illustrate our approach, we have built a MD-SPL where the products are small applications used in programming computers teaching.

***Key Words*—Model Driven Architecture, Variability, Software Product Lines, and Model Transformation**

I. INTRODUCTION

THE latest advances in information technologies, as well as, the constant change in business requirements have transformed software development into a complex task. To face this reality, diverse proposals have arisen, always aiming at producing higher software quality and better productivity of development teams. The principle of these proposals is the reusability of software using compositional and/or generative approaches [1]. The compositional approaches deal with construction of systems by using the join of components that are in a common repository. The generative approaches focus on the reusability of processes and components.

To reuse components to develop applications requires a product family development approach, instead of an independent applications construction approach [2]. A software product family (SPL) is defined as *"a set of software systems that satisfies the specific needs for a particular segment of the market"* [3]. In a SPLs development approach, we can create new products by reusing a set of components called core assets. The main core asset of a SPL is its

architecture. Other kinds of assets include requirement specifications, design models, or software components, among others. Core assets are created during the domain engineering process whereas applications are generated from core assets during the engineering application process [4, 5].

Although products of the same line satisfy specific needs in a given domain, they differ in the set of functionalities for each particular implementation. Thus, all the products of a SPL provide a set of common functionalities, but each product differs from the others in the set of optional (variable) functionalities that it implements. The functional difference between products of a line is known as the SPL variability [5].

Variability management in a SPL can be defined as the set of activities and assets needed to: (1) express common and variable characteristics of a SPL, and (2) construct (compose and/or generate) applications which include common characteristics, and a subset of possible variable characteristics. In a SPL, the products characteristics are directly related with functionality that these products provide.

Currently, feature models are a standard de facto used as a mechanism to support variability management. Feature models are core assets to express common and variable characteristics of a product family. In feature models, variability is represented with features that can be selected in different ways for each product [6]. Thus, starting from a selection of a set of features, and using the core assets, applications with common are variable characteristics are constructed.

On the other hand, an approach of generative reusability is the Model Driven Architecture (MDA) [7]. Unlike other paradigms that use models just like representation, documentation and communication elements, in MDA models play a key role and are first-class elements in application development. The central idea of MDA is to construct business domain models, independently of inherent characteristics of technological platforms. Domain models are then transformed into new models that include characteristics of a specific technological platform through transformation rules.

Models are created using concepts defined in a metamodel. A metamodel represents the concepts, relationships, and semantics of a domain. Hence, metamodels are called also domain models [8]. The relation between a model and its metamodel is defined as a conformity relation, that is, a model conforms to its metamodel [9].

Transformation rules are defined using metamodel concepts. A typical transformation rule takes the source model elements, which conform to a source metamodel concept, and transforms them into target model elements, that conform to different target metamodel concepts [10].

The separation of domains and the generative nature of MDA, make it an adequate approach to create SPLs. The basic idea is to create a product of a family by starting from an initial model, and using several automatic transformations until obtaining a final product.

Results of recent investigations have shown how the mixture of MDA and SPLs (MD-SPL) is an approach (compositional and generative) that makes possible the definition of a SPLs creation process [8, 11]. Nevertheless, variability management in a MD-SPL approach remains as a current area of research.

In this paper, we present a proposal to manage variability during the SPLs construction process, using a MD-SPL approach. For this, we have separated the concepts related to a product line in different domains: (1) the business logic domain, (2) the architecture domain, and the technological platform domain. This separation allows us to manage product lines variability in a separated manner for each domain.

We create feature models for each domain to express the SPL common and variable elements. We also define metamodels for each domain. These metamodels are used to construct a wide set of variable models in each domain, and thus we can express the specific domain variability. In addition to the metamodels, we create transformation rules according to the MDA approach.

Typical transformation rules are created at the metamodel level (M2), which implies that one transformation applied to the same source model always generates the same target model. The above implies that in order to create applications with variable characteristics, we need ways to transform the same model into different target models. In our approach, we successfully achieved this by guiding each transformation with the preferences of the user through a set of variable characteristics.

We define three types of transformation rules: (1) base rules, (2) control rules, and (3) specific rules. We use base rules to generate the SPLs commonalities. Control rules to specify which specific rules will be executed according to the preferences of the user. Finally, specific rules generate different variable features. Using specific rules, we are able to generate applications with different functionality according to the features selection in the SPL applications creation process (application engineering). Thus, using metamodels, transformation rules, and feature models, we achieve to extend the scope of a SPL, and to separate the domains diminishing the complexity of creating applications with variable characteristics in each domain. However, the applications that we generate are not complete. Since some functional requirements are not generated as part of the automatic transformation processes, we manually complete the applications during the engineering process. We will illustrate this proposal with a SPL for the Cupi2 project [12]. For the construction process we used GMF [13] like modeling environment, AMW [14] to make weaving of models, and ATL [15] to transform models.

The paper is organized as follows. In Section 2, we present the application context that will allow us to illustrate the proposal on a concrete example. In Section 3 and 4, we present the problems and challenges of variability management in MD-SPLs as well as the solution proposed to solve it. In section 5, we briefly explained our implementation. Section 6 presents a comparison of our work with some related works, and finally we present the conclusions and we outline future

works.

II. Application context

In order to validate our approach we have implemented a SPL for the Cupi2 project [12]. Cupi2 is part of a series of efforts from the software construction group of The University of Los Andes (Colombia) to find new ways to teach/learn computer programming.

In Cupi2, complete examples and exercises are used to illustrate different topics, instead of just little snippets of code. Each example/exercise includes a graphical interface, a set of requirements, a set of unit tests, java code, and documentation. There are 18 levels in total. Each level adds new concepts to the previous ones. Our SPL generates examples for level 7. In this level, among other topics, we deal with ordering and searching algorithms in collections.

A. *Cupi2 examples commonalities*

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 two components: the kernel and the user interface. The kernel component implements the concepts of business logic. The user interface component implements information visualization and interaction between users and kernel components.

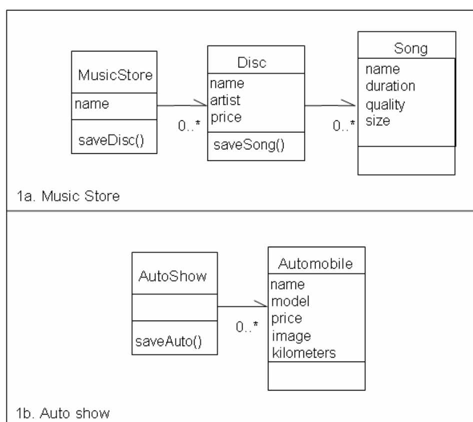


Fig. 1. Music Store and Auto Show models

1) Kernel commonalities: Structures of aggregation can represent and serve to manipulate the kernel concepts and their relationships. In an aggregation structure, there is always a main element that groups the other elements of the kernel. Figures 1a and 1b present business models for two different examples: a Music Store and an Auto Show. In the Music Store model, shown in Figure 1a, a MusicStore assembles a set of Discs and each Disc assembles a set of Songs. The Auto Show model of Figure 1b shows the AutoShow assembling a set of Automobiles.

All the kernel elements have a set of properties and are related to other elements of the kernel. Finally, each kernel element is responsible of persisting its own information

2) User interface component commonalities: To build a graphical user interface (GUI), Cupi2 applications employ a set of elements like 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 communicating the kernel and the GUI by grouping all the views. The *ExtensionView* contains several buttons that students can use to add new functionality to applications as part of the exercise.

B. *Cupi2 Examples Variability*

At the same time that we identify commonalities, we identify variability as well. The variability in Cupi2 examples is related to the algorithms that manage the aggregation structures in the kernel component, and the presentation of the GUI.

1) Kernel variability: The kernel component has variable elements related to data structures, algorithms, and services to persist the different assemblies. The data structures that represent the aggregation structures can be containers of fix or variable size, or can be linear structures such as simple lists or double-linked 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 of elements, to search a particular element, or to order the set of elements, among other services.

Different implementations can be used to persist kernel elements information; for example text files with a special structure or object serialization

2) User interface variability: There is not a single way to represent the kernel elements in terms of GUI elements. The user can select one or several types of views to represent the kernel elements. Such views are: *main view, extension view, set view, search view, information view,* and *aggregation view.*

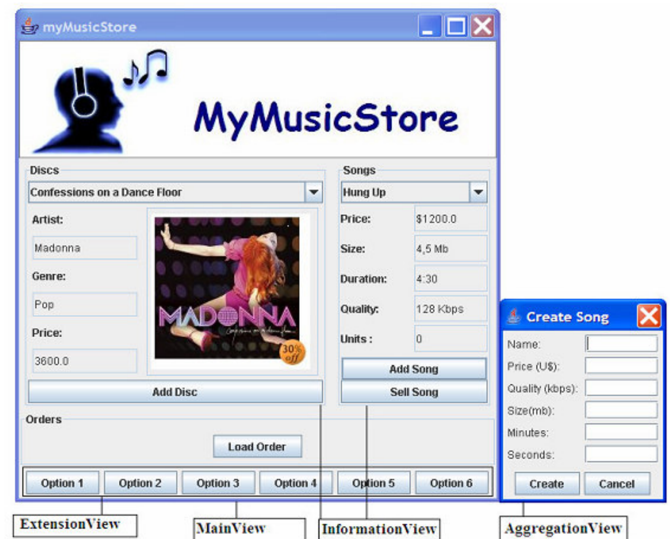


Fig. 2. MusicStore GUI

a) *Main View*: The *MainView* contains the other views

and is responsible for the communication between the kernel and the user interface.

*b) Extension View:*The *ExtensionView* has a set of buttons. Student may associate functionalities to each button.

c) Set View: The *SetView* presents a set of grouping elements using components such as lists or tables.

d) Search View: The *SearchView* is used to introduce the parameters for a search.

e) Information View: The *InformationView* is used to show particular information associated to attributes of the kernel, this information may include images.

f) Aggregation View: The *AggregationView* is used to enter the information of elements that are being created.

In Figure 2, we present a possible GUI configuration with four different types of views for the Music Store example.

III. Domain Engineering Process

The domain engineering process involves the creation of core assets. In this section, we introduce the core assets used in the Cupi2 SPL.

We group concepts in different domains in order to manage variability. These domains are the business logic, the architecture domain, and the technological platform domain. Our strategy follows the MDA approach and it is based on the automatic transformation 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) weaving models and three types of transformation rules. Figure 3 presents the domain-engineering process.

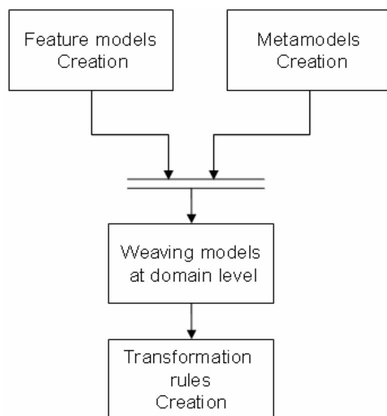


Fig. 3. Domain engineering process

A. Feature Models

Feature models are a standard de facto used to represent common and different characteristics of members from a product line [16]. The features can be mandatory, optional, or alternative. Mandatory features are common to every member of the software product line; some members require optional features. Two or more features are alternatives to each other, when the user can choose only one of them.

A feature diagram is a tree which nodes represent features [2]. There are two kinds of nodes: grouped nodes and solitary nodes. A solitary node is not contained in a grouped node. Solitary nodes represent mandatory or optional features. Grouped nodes group mandatory and optional features, or alternative features.

In our approach, feature models describe common and different characteristics of applications in target domains. Cupi2 SPL has two target domains: architecture and technological domain. We develop a features model for each domain.

Figure 4 presents a subset of features for the architecture domain. The model represents the user interface characteristics (section 2). The *GUI* node groups a *MainView* (mandatory feature) and *SetView*, *AddView* and *InformationView* (optional features).

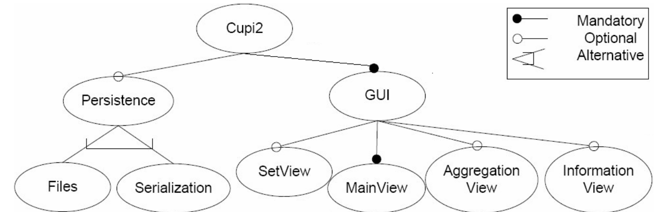


Fig. 4. Cupi2 architecture features

Figure 5 presents a subset of features for the technological domain (Java). This model has a Container node, which groups the ArrayList and Vector nodes (alternative features).

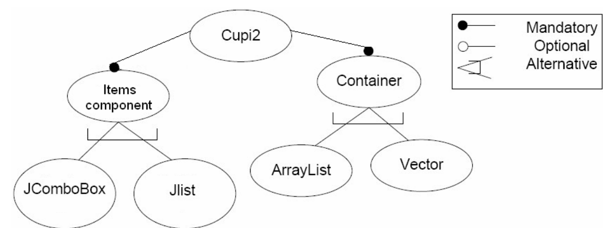


Fig. 5. Cupi2 technological features

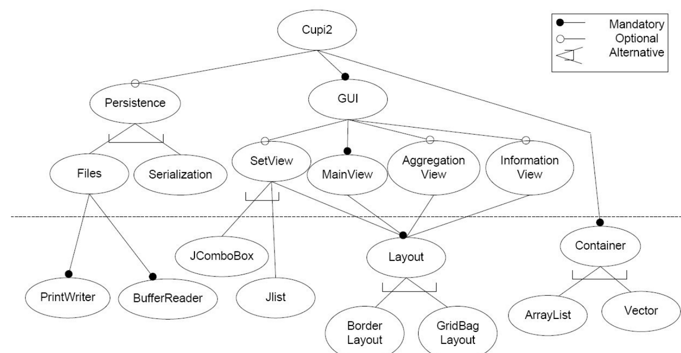


Fig. 6. Cupi2 features model

Figure 6 presents the relationship between architectural and technological feature models. The relationship between

features nodes implies that the nodes selected in the architectural feature model constrain the selection the user can make in the platform feature model. For example, the selection of *JCombobox* or *JList* features can be done only if the *SetView* feature has been selected before.

B. *Metamodels*

As stated in section 1, in our approach metamodels represent the concepts, relationships, and semantics of the domain of applications.

In a similar way as the feature models express common and variable characteristics of a set of applications, metamodels also allow the expression of such characteristics. This is achieved by abstracting the domain concepts, their properties, and their relationships. Each model that conforms to a metamodel represents a particular instance of the domain, with a set of specific characteristics.

To create the Cupi2 project SPL, we have designed three different metamodels as part of the core assets. These metamodels refer to business logic, architecture, and technological platform. Each metamodel includes abstract concepts of a particular domain. Thus, business metamodel only includes the essential concepts of the problem to solve in the Cupi2 examples. Architecture metamodel includes refined concepts of the business metamodel, along with the concepts of the architecture including the user interface. Finally, the technological platform metamodel includes refined concepts of the architecture metamodel along with its own concepts of the language such as for example packages, classes, methods and attributes.

1) *Business logic metamodel:* The business logic metamodel presented in Figure 7 represents abstract concepts from the Cupi2 examples of levels 7 and 8. In general terms, these examples describe a set of elements related to each other through aggregation structures. For instance, in the Auto show example, we say that an *AutoShow* element groups a set of *Automobile*s (Figure 8). In addition to that, each element may have a set of attributes that characterize it. For example, it is possible that the *Automobile* element has an attribute that represents the model; this attribute can be used later to create a service that orders all the instances of *Automobile* per model.

Figure 7 presents the concepts Container and Element, and the aggregation relationship between them. The AttributeCupi2 is also presented, as well as all the attributes that characterize properly this business concept according to the requirements of the applications being modeled. For example, the attribute isComparable in the meta-class AttributeCupi2, indicates that such element could be compared to other elements. This information is used in the transformation process and may lead to the generation of algorithms to order a set of AttributeCupi2 instances or to look for a single AttributeCupi2 element.

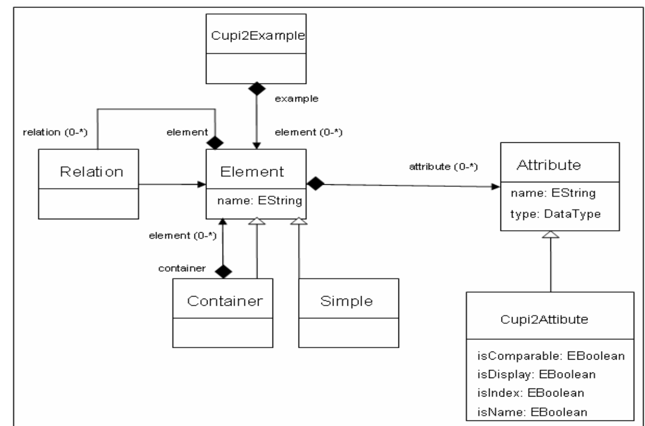


Fig. 7. Business logic metamodel

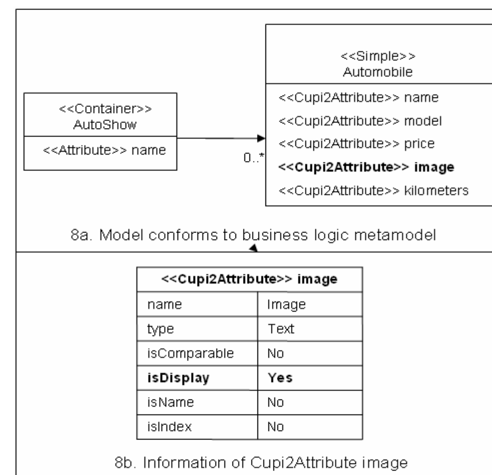


Fig. 8. Auto show business logic model

Models that conform to the business metamodel, use its concepts to express the features of a single instance of the domain they represent. In Figure 8a an example of a model for the Auto Show example is shown. In this model representation as well as in the rest of models in the reminder of this paper, we use stereotypes as *<<Container>>, <<Simple>>, <<Attribute>> and <<AttributeCupi2>>* to indicate the "*conforms to*" relationship between the elements of the model and its counterpart in the metamodel. The *AutoShow* element conforms to *Container,* which, according to the metamodel definition, indicates that it may group a set of other *Container* or *Simple* elements.

In this example, the grouped element is the Automobile. Each Automobile contains a set of Attribute and AttributeCupi2 elements. Figure 8b presents a detailed view of each of these attributes.

2) *Architecture Metamodel:* The architecture metamodel is related to design concepts of Cupi2 examples; this metamodel does not include details about implementation on a specific technological platform. To simplify its representation, we have divided it in two different metamodels. The first one includes the concepts related to the business logic and the second one deals with the concepts of graphic user interface. The business logic architecture metamodel introduces the concept of

Service. Services are necessary to manipulate the data structures of the examples. On the other hand, the graphic user interface metamodel includes interaction concepts and elements that will be part of the graphical user interface of the generated application. A model that conforms to the architecture metamodel has all the structural and graphical elements of a real application. From this point, it should be possible to specify a technological platform and generate the source code.

In Figure 9, we present a simplified version of the user interface architecture metamodel. This metamodel aims for the generalization of concepts found in the different view types described in section 2. The main element refers to V*iew. A View* can simultaneously have a set of *Component*s and a set of *Service*s. The *Component*s are specialized in two types, *Interaction* and *Visualization*. The *Interaction Component*s are used in a GUI to call some type of functionality. Some examples of *Interaction Component*s can be buttons, lists, or drop-down lists. On the other hand, the *Visualization Component*s are used to show information to the user. Some examples of these *Component*s are text labels, and message boxes.

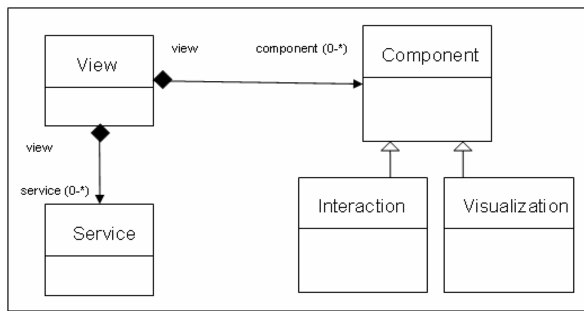


Fig. 9. GUI architecture metamodel

Figure 10 presents a model that conforms to the GUI architecture metamodel. In the same way as in figure 8, we use stereotypes to show the relation of *conformity* of the elements of the model with the metamodel.

In this case there are four different elements conform to the *View* element of the metamodel. Each *View* has a set of components according to its specific responsibilities. The *AutoShowMainView* element is the main view and contains all other views. The *AutoSetView* is in charge of showing a group of automobiles contained by the data structure of the application. This view has an *Interaction Component* of type *List* to show the set of automobiles. The *orderByModel* button is associated with the services of ordering the list of automobiles. The *AutoSearchView* offers the possibility of look for a automobile using the attribute *name*. Finally, the *AutoInformationView* has a set of *Label* elements to visualize the automobile attributes.

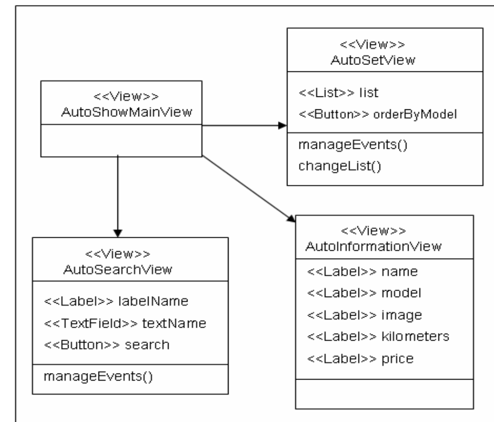


Fig. 10. Auto show GUI model

C. Weaving model at domain level and transformations rules

Our strategy uses automatic transformation of models until obtaining executable applications.

Transformations 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. There are three types of rules: (1) base rules, (2) control rules, and (3) specific rules. In order to guide the creation of these transformation rules, we create weaving model at domain level.

Weaving models link metamodel concepts to feature nodes. Each link means that one source metamodel concept can be transformed to obtain the target feature. Using the weaving models at domain level, 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 for the variability.

Figure 11 presents a weaving model between the business logic metamodel and the feature model of the architecture domain. In Figure 11, the link between Container and the mandatory feature MainView indicates that a main view is always created for Container 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.

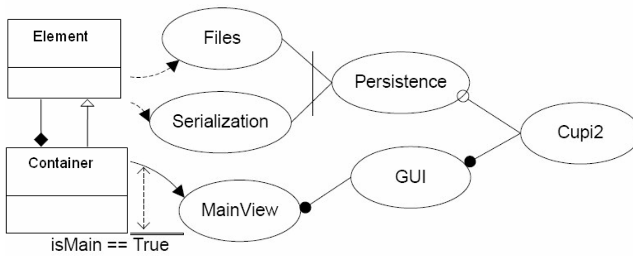


Fig. 11. Weaving model between the business logic metamodel and the architecture features

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 one.

Listing 1 presents an example for each type of rule. The *mainView* is a base rule. Line 3 shows the source pattern and the line 5 shows the target pattern. The *mainView* rule transforms the elements conform to *Container,* without container*,* into *View*. For example, for the Music store application, a *View* is created from the *MusicStore* element (conform to *Container*). For the Auto Show application, a *View* is created from the *AutoShow* element.

```
1.   rule mainView{
2.     from
3.       g : Container (g.container.oclIsUndefined())
4.    to
5.       v : View
6.   }

7.   rule setView{
8.     from
9.       l : Link (l.feature == 'SetView')
10.    using {
11.      e : Element = l.getElement()
12.    to
13.      v : View
14.    do{
15.       addComponent(v, #list);
16.     }
17.   }

18.   rule addComponent(v : View, t: Type){
19.    to
20.        c:Visualization(
21.          type <- t
22.        )
23.    do{
24.        v.attribute.add(c);
25.    }
26.  }
```

List 1. Base, control, and specific transformation rules

The setView, in line 7, is a control rule. The declarative section is in the lines 8 to 13 and the imperative section is in the lines 14-16. The setView rule searches in the weaving model the elements that have the feature SetView associated (line 9). For those elements, a View is created (line 13) and the addComponent imperative rule is called (line 15). 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 lines 23-25. When this rule is called from the control rule, a Visualization component is created (lines 19-22), the parameter type is assigned to this component, in this case List (line 21), and the created component is added to the previous created View (line 24).

IV. APPLICATION ENGINEERING

Application engineering process generates a specific product from the core assets [4, 5].

As we said previously, our strategy uses an incremental transformation of models until obtaining applications. Thus, we start from a business logic model, we transform it into an architectural model, after that we transform it into a model in the technological platform domain and finally we generate source code from it. Figure 12 presents the application engineering process.

A. *Business logic model*

The first step is to create the business logic model. Business logic model represents the problem description. Figure 8a shows a business logic model.

B. *Weaving model at application level*

User selects the features that he wants before the execution of each transformation. He makes the selection by associating (source) model elements and (target) feature nodes. We use weaving model to create the associations.

Weaving model at domain level constrains weaving model at application level. For example, the weaving model (showed in Figure 11) contains a link between *Element*, and the *Serialization* and *Files* feature nodes. Thus, weaving model at application level can contain a link between an element (conforms to *Element*) and *Serialization* or *Files* feature. Figure 13 presents a weaving model at application level.

Figure 13 shows links between business model elements and architecture features. For instance, one link associates *Disc* element and *SetView* feature. It generates a *SetView* which groups discs by using a list. Another link associates *MusicStore* element and *Serialization* feature. Link between *MusicStore* element and *Files* feature is not possible because *Serialization* and *Files* are alternative features.

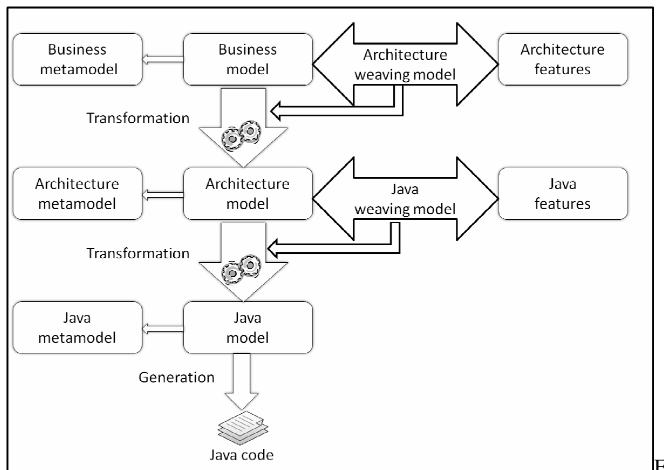
User should create two weaving models to generate a Cupi2 example. First, he weaves business model elements and architecture features. Later, he weaves architecture model elements and technological features.

C. *Transformation execution*

After the weaving at application level, a transformation is executed.

A transformation has two inputs (source model and weaving model) and one output (target model). Transformation applies base, control, or specific rules. For example, for Music store application, business to architecture transformation receives the business model (showed in Figure 1a) and weaving model (showed in Figure 13). When transformation is executed, *mainView* and *setView* rules are executed too; *mainView* is

always executed, *setView* rule is executed because we weave *Disc* (conforms to *Element*) and *setView* feature.



Fi

Fig. 12. Weaving model between business model and architecture features.

V. IMPLEMENTATION

We used feature models, metamodels, and transformation rules like core assets in the SPL application engineering process to create specific example applications such as the Music store, the AutoShow, and others. We created feature models for the architecture domain and the technological platform. We created business logic, architecture, and technological platform metamodels. Finally, we created base, control, and specific transformation rules. These transformation rules transform business logic models to architecture models, and architecture models to technological platform models.

In the application creation process, the developer creates a model of the business logic. Then, the model is weaved with the architecture feature model and transformed to obtain an architecture model. The generated architecture model is weaved with the technological platform feature model so it can be transformed into a platform dependent model. Finally, we apply a transformation based on templates to generate the source code (see Figure 12).

For the implementation of our solution, we have selected a suite of Eclipse tools (plug-ins) [17]. described in the next sections:

- *A model and metamodel manager*: this tool allows us to create metamodels and models that are serialized in XMI format following the MOF standards proposed by the OMG. This tool is EMF [18]. EMF implements EMOF (Essential MOF), which is a subset of MOF.
- *A graphical modeling tool*: Due to the complexity in metamodels expression, we require a tool to create metamodels in a graphical way. We have chosen GMF, since it allows us to define metamodels graphically and to generate plug-ins to create models that conform to these metamodels [13].
- *A model-to-model transformation language*: we have selected ATL as transformation language. This is a mixed language (declarative and imperative) compatible with models expressed using EMF [18]. Even though ATL is not a full QVT implementation (Query, View, Transformations), the standard proposed by the OMG to make queries and transformations on the models and metamodels, it has a very good and consistent implementation.
- *A tool for weaving models*: we have selected AMW [14] as tool for mapping or link elements of two different models, and to generate a model with the information of these links. The AMW models are also compatible with models expressed with EMF.
- *A model to code transformation language*: We use Acceleo [19] as transformation language. This tool specializes in the generation of text files (code, XML, documentation) starting from models. Using Acceleo we can generate the source code based on templates and models expressed with EMF.

VI. RELATED WORK

Proposals for the SPL variability management focus on managing variability of members in a product family. The differences among these proposals and ours reside in the core assets that are used to express the variability, and to create (variable) SPL applications during the application engineering process. In this section, we have tried to put together different proposals for managing variability in SPLs. However, our goal is not to present all the state-of-the-art of the subject, some proposals could not be referenced here.

The FODA method [16] was introduced as a strategy to express variable functionality in the requirements engineering process through the use of feature models. The FORM [20] proposal complements the FODA proposal to express variable functionality in design applications process, and prescribes how feature models can be used as a basis to develop domain architectures and components for reusability. In FORM, the authors propose to organize the features in agreement with non-functional requirements, and use object- oriented components as core assets to create SPLs applications.

AHEAD [21] is a proposal of an architecture model for object oriented programming, and a base for compositional programming on large scale. In AHEAD (Algebraic Hierarchical for Equations Application Design), the feature models are used to express variability. The core assets to create SPL applications are fragments of classes and methods. The features are associated with these assets, and they are composed by means of algebraic equations to create SPL applications.

In [22] a proposal with classes and aspects like core assets is described. To express variability, the authors propose the elaboration of feature models. For the development of the SPL they propose: (1) to design a flexible architecture applying patterns, (2) to design aspects regarding the variable features,

and (3) to compose the aspects and business classes.

As it can be inferred from these proposals, feature models are a *standard de facto* to model the SPLs variability. Nevertheless, there are other proposals that use models and express variability. In [23] an orthogonal variability model (OVM) is proposed to reduce the feature models complexity. The variability models are constructed conforming to a general metamodel that defines the variability concepts. In [24], there is also a process based on UML for the expression of variability and the definition of mechanisms that allow its implementation.

Recent works demonstrate the advantages of Model Driven Engineering (MDE) in SPLs variability management. In [2] the author presents, from a global perspective, the basic concepts of the Generative Software Development (GSD) approach. This approach aims at developing product families by automating the family member's creation from specifications written in specific domain languages. These languages can be defined using meta-modelling.

In [25] MDA is presented as a mechanism for variability expression that makes possible to postpone the decision-making task on the feature model. In [11] the MDA abstraction levels are employed as a mechanism to express variability in a separated way for each domain that corresponds to an abstraction. The SPL applications are created composing elements of a common framework. The approach described in [8] studies the reusability of core assets to extend variability scope by using metamodels with different abstraction levels. The composition of applications is achieved by mapping business abstract concepts with the low-level components. These MD-SPL proposals do not use model transformations like mechanism for SPL applications generation.

In our proposal, we use the following core assets to manage the variability: (1) feature models, (2) metamodels, and (3) transformation rules. Like in some of the previous proposals, we use feature models to express the SPL applications variability. Nevertheless, in our strategy to generate applications, we create different feature models to express in a separated way, the product line variability for each specific domain. In addition, as a complementary strategy to the feature models, we use metamodels to express variability of different domains. In consequence, we successful extend the SPL scope.

Our strategy to generate applications is mainly generative. We automatically generate the applications with variable characteristics of the SPL using transformation rules on models. During each transformation process it is possible to select features. Thus, we divide the decision-making task on the feature models, postponing until the last moment the decisions related to technology and platform.

VII. CONCLUSIONS

In this paper, we have presented a proposal to manage variability during the SPLs construction process using a MD-SPL approach. For the core assets creation, we separate concepts related to a SPL in different domains. This separation allows us to extend the SPL scope, managing the variability at level of concepts of each domain in an independent way. Our strategy for the applications creation uses automatic transformation of an initial business model into a target architecture model; then, we start from the architecture model, and we transform it into a target technological platform model; finally, we transform the technological platform model into source code.

In our approach, we only build the initial business logic model manually. From that initial model, we automatically generate new and more refined models and finally, we generate java code from such models. Having a process separated in several stages allows us to construct simpler and more flexible transformation rules than processes to generate in only one-step a complete application. Since the applications that we generate are not complete, not all the functional requirements are created, part of the future work refers to represent the concepts for all the functional requirements in the business logic, and to construct the transformations of these concepts.

In order to guide the generation of applications with different (variable) characteristics in each domain, we create metamodels, feature models, weaving models and three types different of transformation rules: (1) base, (2) control, and (3) specific rules. The feature models allow us to express and to select the desired characteristics of an application in a target specific domain. The metamodels allow us to create large set of (variable) models in a particular domain extending the SPL scope. The weaving between metamodels and feature models makes possible the identification of the transformation rules needed to generate common and variable characteristics of the applications. The transformation base rules allow us to create applications with common characteristics of the line. The specific and control rules allow us to create applications with variable characteristics of the product line. Finally, the weaving models between source models and target domain feature models allow the automatic rules execution to generate models of variable applications in each domain. Thus, using these different core assets we can manage the variability not only at application level, but also at the domain level. This means that we can generate applications that vary not just in concrete functionalities at application level, but also in concepts of the different identified domains.

Currently we are validating our approach with the implementation of the SPL to generate different applications. As future work, we want to extend the SPL to include different Cupi2 levels. By doing that we will be able to validate the concepts of this proposal in domains where the business logic handles a higher number of concepts.

The construction of an eclipse Plug-in that allows the creation of the weaving models, the selection of features in each domain, and the automatic applications generation is part of the current work.

One of these topics is the need to explore new alternatives for domains separation, and modeling the concept for each

domain. A close related future work refers to the conceptual expression of different domains; it is necessary to work on the expression and transformation of concepts relative to functional requirements. It is also necessary to work on the design and development of modular rules starting from a better-defined transformation pattern. To complement the transformation processes, and in general, the generation processes, the traceability management is a complete field to explore.

REFERENCES

[1] J. Sametinger, *Software engineering with reusable components*, New York: Springer, 1997.
[2] K. Czarnecki, "Overview of Generative Software Development." pp. 313–328.
[3] "Software Product Lines," December, 2006; [Online]. Available: http://www.sei.cmu.edu/productlines/.
[4] P. Clements, and L. Northrop, *Software Product Lines: Practices and Patterns*: Addison Wesley Professional, 2001.
[5] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*: Springer, 2005.
[6] J. Lee, and D. Muthig, "Feature-oriented variability management in product line engineering," *Communications of the ACM,* vol. 49, no. 12, pp. 55 - 59, 2006.
[7] J. Mukerji, and J. Miller, "MDA Guide," 2003.
[8] J. Estublier, and G. Vega, "Reuse and variability in large software applications," in 13th ACM SIGSOFT Lisbon, Portugal 2005.
[9] J. Bézivin, "On the Unification Power of Models," *Software and Systems Modeling,* vol. 4, no. 2, pp. 171-188, 2005.
[10] O. M. G. OMG, "MOF 2.0 Query/Views/Transformations RFP," 2002].
[11] A. L. Santos, A. Lopes, and K. Koskimies, "An MDA Approach to Variability Management in Product-Line Engineering ".
[12] "Proyecto CUPI2," Diciembre, 2006; [Online].
[13] "Graphical Modeling Framework " Access 2006; [Online]. Available: http://www.eclipse.org/gmf/.
[14] "AMW Home Page," Access 2006; [Online]. Available: http://www.eclipse.org/gmt/amw/.
[15] "ATL Home Page," Access 2006; [Online]. Available: http://www.eclipse.org/gmt/atl/.
[16] K. Kang, S. Cohen, J. Hess *et al.*, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Estados Unidos 1990.
[17] "Eclipse," Access 2006; [Online]. Available: http://www.eclipse.org/.
[18] "Eclipse Modeling Framework," Access 2006; [Online]. Available: http://www.eclipse.org/emf/.
[19] "Acceleo," Access 2006; [Online]. Available: http://www.acceleo.org.
[20] K. C. Kang, S. Kim, J. Lee *et al.*, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering* vol. 5, pp. 143 - 168, 1998.
[21] D. Batory, "Feature-oriented programming and the AHEAD tool suite."
[22] V. Alves, A. Dantas, and P. Borba., "AOP-Driven Variability in Product Lines of Pervasive Computing Applications," in Second International Generative Programming and Component Engineering Conference (GPCE'03), Erfurt, Germany, 2003.
[23] K. Pohl, F. van der Linden, and A. Metzger, "Software Product Line Variability Management," *International Software Product Line Conference*, 2006.
[24] E. A. d. Oliveira, I. M. S. Gimenes, E. H. M. Huzita *et al.*, "A variability management process for software product lines." pp. 225 - 241
[25] S. Deelstra, M. Sinnema, J. v. Gurp *et al.*, "Model Driven Architecture as Approach to Manage Variability in Software Product Families."

Kelly Garcés is a master graduate from University of Los Andes, Colombia. Currently she is doing an internship at the Ecole des Mines de Nantes, France. Her interests are Model Driven Engineering and the improvement of software development processes.

Carlos Parra is a master graduate from University of Los Andes, currently he is doing an internship at the University of Science and Technology in Lille, France. His interests are Model Driven Engineering and Context Aware Software.

Hugo Arboleda is a Ph.D. student at University of Los Andes in Colombia in the Software Construction group; and at Ecole des Mines Nantes, France, in the OBASCO group. He received a M.Sc. in Computing and System Engineering from Los Andes University in 2004. His interests are Model Driven Engineering and Software Product Line Engineering.

Andres Yie is an Ph.D. student from the University of Los Andes and Vrije Universiteit Brussel. He received a M.Sc. in Computing and System Engineering from Los Andes University in 2006. His interests are Model Driven Development and Software Product Lines looking for semi-automatic strategies on software development.

Rubby Casallas is an Associate professor in the Department of Systems and Computing Engineering, University of Los Andes, Bogotá, Colombia. She received a Ph.D. in Informatics from the University of Grenoble, France in 1996. Currently she is the coordinator of the Software Construction group at the University of Los Andes. Her interests are Software Engineering Education and Software Product Lines.