Published at Software and Service Variability Management Workshop. Concept Models and Tools.
Helsinki, Finland, April 19-20 2007.

Variations in Model-Based Composition of Domains

Anca Daniela Ionita[1], Jacky Estublier[2], German Vega[2]

[1]Automatic Control and Computers Faculty, Univ. "Politehnica" of Bucharest,
Spl.Independentei 313, 060042, Bucharest, Romania Anca.Ionita@mag.pub.ro
[2]LIG-IMAG, 220, rue de la Chimie BP53 38041 Grenoble Cedex 9, France
{Jacky.Estublier, German.Vega}@imag.fr

Abstract. As model driven engineering increases the level of abstraction, there are more possibilities to hide complexity and to introduce variability points. Variations and composition, which are usually complementary approaches, may be merged inside models and complement each other. The "domains", presented in this paper represent a coarse granularity reuse units. Domain variability is defined by models and the way domains are composed is also defined by a (composition) model. Therefore, variations are found at two levels, (1) inside domains (by choosing the appropriate model) and (2) in the composition mechanism itself (by defining the composition semantics).

Keywords: Model Driven Engineering (MDE), variability, composition, reuse

1 Introduction

For managing diversity, variation and composition are often considered two independent "schools of thought", but merging them has been identified as a necessity [1]. Variation, in product family approaches, supposes a top-down approach, based on an architecture where the elements, common to the whole family, and the "features" specific to each family member, are clearly identified. While planned variations are easy to perform (feature selection), unplanned evolution is difficult to address. Conversely, composition supposes a bottom-up approach, where existing units are selected and glued, in order to form an application. Composition is not limited to putting together components that perfectly fit together, but must also solve functional and non functional mismatches. This paper investigates the links between variation and composition, supported by Model Driven Engineering [2].

Composition may be seen as a variation mechanism because it allows building different combinations which constitute variations, and hierarchical composition [3] increases the variability potential. The current composition approaches use a rigid composition mechanism, which reduces both the reuse potential and the range of possible assemblies. Generally, variation only consists in changing method call parameters and/or in selecting the classes/components implementing the required interfaces. The composition logic and the variability are spread among components and cannot be changed without changing their code. To enhance flexibility, reusability and evolutivity, the control flow should not be the scattered in the components, but centralised, like in orchestration [4].

To improve both variation and composition, MDE raises the level of abstraction of the applications, allowing abstract composition mechanisms independent from

technical details. In an MDE approach, the elements to compose are models, and composition means more than selecting and gluing pieces of code. A powerful model composition mechanism, which includes variability capabilities, is needed. Composition can be used to create large applications pertaining to product populations [5], but new concepts and mechanisms are required to support this complexity.

The paper discusses a new model-based composition mechanism, tailored for large units of reuse called "domains". Each domain is associated with a DSL (a metamodel) and an application in the domain is defined as a model written in this DSL (conform to the metamodel). Composing two applications (A and B) pertaining to two domains reduces to compose their models (MA and MB). First, our approach composes the two DSLs (DSL_A and DLS_B) through the definition of new relationships between them and obtains a new DSL (DSL_AB). This (meta) composition may include some variability. Then, MA and MB are composed, while the composition model is expressed in DSL_AB, *leaving MA and MB unchanged*.

A development environment, called Mélusine has been realised and a correspondent methodology [7] has been defined for supporting this approach. Chapter 2 describes the variability introduced by the domain architecture, while chapter 3 presents the variation points related to domain composition through relationships. Both of them are exemplified on real examples used and reused in industrial applications.

2 Variations and Composition with Domains

2.1 Variability inside Domains

The *domains* presented in this paper have been designed to reuse heterogeneous components that do not know each other. The composition model is explicit, according to the domain specificity and to the application variability. The layered architecture (see 3.1.) allows the definition of variation points on multiple levels.

Domains are fully independent and autonomous: they do not have dependencies of any kind. A domain has a *domain specific language* and can interpret models conforming to it (see Fig. 1). This DSL gives an abstract and stable description of the problem to solve and also gives the possibility for the domain experts to deal the concepts, without knowing technical details. An example of DSL is presented in Fig.1 for Product domain, used as a basic versioning system for various products, characterised by types and attributes.

After defining the domain DSL, a *DSL interpreter* is realized by mapping the abstract architecture on concrete components, which are called *tools* in our approach. The mapping is not done directly, but in terms of a *feature model* that captures functional and non-functional properties, which are optional and may be maintained independently from the DSL.

Once the domain and its interpreter are available, creating an application simply stands in choosing the variation points at three levels: (V1) defining the application model conforming to the DSL, (V2) selecting the features and (V3) selecting the

components (tools). Thus, application development becomes a simple task that can be performed by non experts.

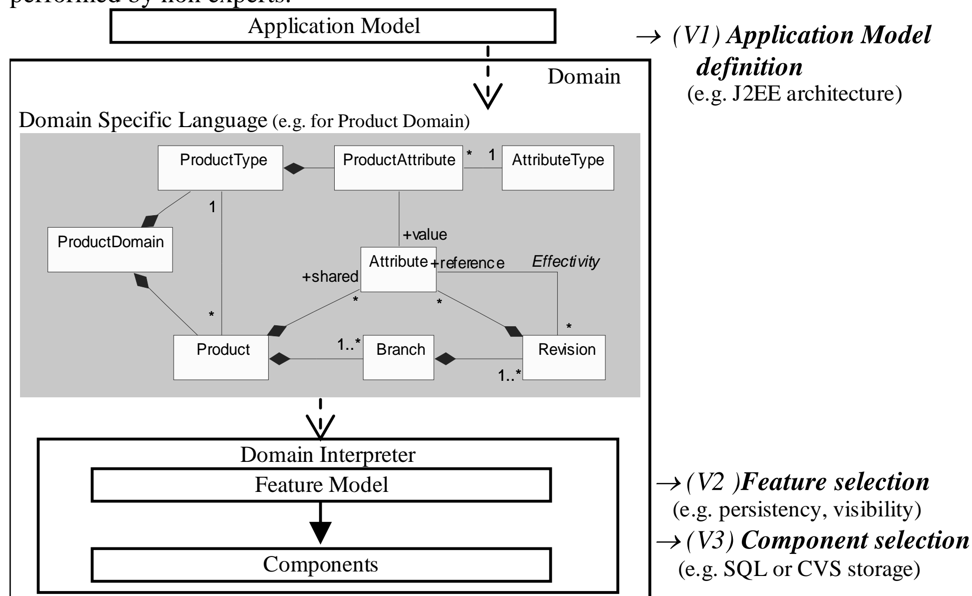

Fig. 1. Variation points inside the domain architecture

2.1.1 Application Model Definition

For a particular application, one has to define a model conforming to this DSL, which is then understood by the domain interpreter. As the model is often structural only, the application development does not need any programming [7], so it may be performed directly by domain experts and not necessarily by software engineers. E.g., if Product domain is used to version software artefacts conforming to J2EE architecture, a product type may correspond to the Servlet concept.

2.1.2 Feature selection

The DSL incorporates the variability regarding the domain concepts, but there are other variations, regarding behaviour or non-functional properties, which are not related to a single concept. For this reason, the DSL, which is defined a priori, is complemented by a feature model, which may be identified and incorporated in the system a posteriori. As this feature model does not try to grasp all the variability, it is rather simple and does not need the generality of the feature models from product lines [8]. Features are optional and may be selected through the wizard given by Mélusine environment. Examples of features for Product domain are:

- *persistency* – related to the objects instantiated from Product, Branch and Revision;
- *visibility* – related to visualizing products and revisions stored in ProductDomain.

2.1.3 Component selection

The domain realization finally stands in mapping the abstract features to concrete components, which contain most of the implementation code. It has been a purpose of this approach to be able to reuse non-homogenous tools, using wrappers that adapt them to abstract components. The tools for implementing an abstract feature may be chosen through a wizard. For instance, the *persistency* feature may be mapped on components having different implementations, based either on SQL storage, or on the repository of a versioning system like Subversion or CVS.

2.2 Variation through domain composition

Domains are by nature narrow in scope, so applications usually span over several domains. Therefore, a domain composition mechanism is necessary, such as to:

- "match" any pair of domains, irrespective of their DSLs;
- keep unchanged the original implementation.

Being a (large) unit of reuse, a domain has an "interface", or a visible part, which is actually its DSL, expressed as classes and relationships. This DSL is used to perform composition between domains at a high level of abstraction, irrespective of features and components used at lower levels. Domain matching is performed by creating relationships between concepts, to support flexible interactions. The goal of preserving the code was obtained through aspect oriented programming (AOP).

The composition mechanism is exemplified on *RichProduct*, a composite domain reused for various purposes, from document management to software configuration management. It is obtained by composing the autonomous domains *Product* and *Document*, while preserving their original implementation. The idea behind the *RichProduct* domain was to have a versioning system for complex, "rich" elements, which are both characterized by some attributes (managed by *Product* domain) and by supplementary information stored in one or more files (managed by *Document* domain).

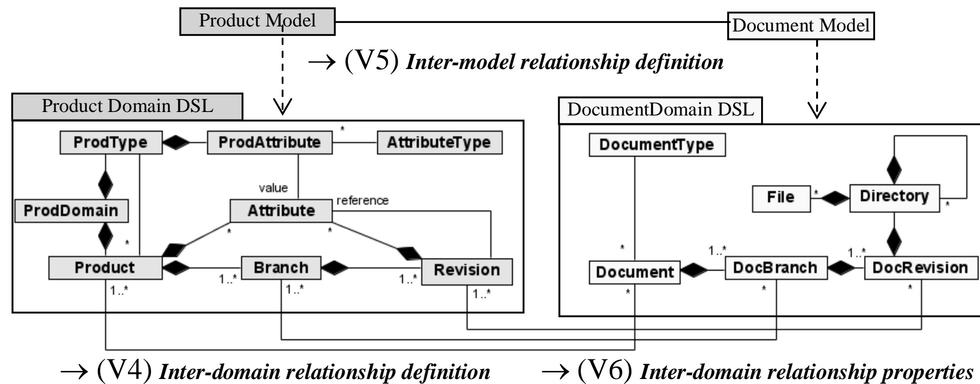

Fig.2. Variation points in the domain composition mechanism

2.2.1 Inter-domain relationship definition

The composite domain DSL includes the subdomain DSLs (see Fig. 2) plus relationships that either *establish interactions* characteristic to the new composite domain, or link "the same" concept found in both domains. These relationships are called *inter-domain relationships*. (e.g. Product – Document or Revision – DocRevision, exemplifying the variation point V4 in from Fig.2)

Inter-domain relationships are not modifying the code of the original domains; links and interactions are implemented in a non-invasive way, using AOP.

2.2.2 Inter-model relationship definition

An application pertaining to a composite domain is characterized by several models, one for each of the composed domains. For developing a new application, the application designer should select a model from each domain and should define the relationships between these models, in conformity with the previously described inter-domain relationships. These inter-model relationships (variation point V5 from Fig. 2) may be established either *automatically* (if there are enough criteria for matching the model elements) or *manually*, through another Mélusine wizard (methodological details were given in [7]).

2.2.3 Inter-domain relationship properties

The inter-domain relationships in composite domains are characterised by several properties, which also constitute the variation points of our composition mechanism (see V6 from Fig.2). Some characteristic attributes are given below:

a) ***Destination Life Cycle Management***:
   - *Source Independent* - if objects from the source and destination classes are created independently and then related by an implicit or an explicit selection; this is similar to an association in UML;
   - *Source Dependent* - if the life cycle of the objects instantiated from the destination class is managed by the objects instantiated from the source class.

Source Dependent relationships are similar to UML composition but the implementation is different from a composition between classes belonging to the same model.

In our example, the documents life cycle, with branches and revisions, is under the control of products, with their respective revision tree. Thus, the destination life cycle is dependent on the source one.

b) ***Multiplicity*** – an object from the source class may be linked to one or more objects from the destination class and vice-versa; the link may be optional or not.

Multiplicity may vary with respect to the composition policy and strongly depends on the context in which the composed domains will be used. Let us take as example the multiplicity choice for the Revision class related to DocRevision. What happens when the product attributes change, while the document content remains the same? If the multiplicity is 1..*, a document revision may be linked to more product revisions.

c) ***Link Creation Moment*** - represents the moment at which the source and destination instances are linked:
   - *Early, at instantiation* - means creating the link immediately after the instantiation of the source;

- *Late, at navigation* - means creating the link when navigation requires the destination object;

In *RichProduct,* a document is instantiated immediately after creating a product and linked to it. So happens to branches and revisions also.

d) ***Persistence*** – if the link is persistent and must be recreated in case of failure;

e) ***Captures*** - the signature of methods that have to be captured for supporting interactions across the relationship.

Mélusine environment includes a user interface that asks the composite domain developer to make the right choice for these properties. The code for implementing the above properties (like establishing and navigating the links, persistency, or destination object creation) is generated automatically in the form of AspectJ extensions. Other complex interactions between the linked objects should however be manually coded, on the basis of the skeleton generated for the captures.

3 Conclusion

The increase of size, complexity and evolution requires more flexible composition mechanisms, such as to manage variation points inside the units of reuse, but also in the composition mechanisms itself. This may be obtained through model-based composition, at a high level of abstraction. This paper presents such a composition mechanism for high granularity units of reuse called domains. Any pair of domains may be composed by establishing inter-domain relationships, whose properties represent variation points of the composition.

Acknowledgments. The work of Anca Daniela Ionita was supported by a FP6, Marie Curie – EIF fellowship at LIG-IMAG and by MEC-CNCSIS A37 grant.

References

1. van Ommering, R., Bosch, J.: Widening the Scope of Software Product Lines – From Variation to Composition, G. Chastek (Ed.): SPLC2 2002, LNCS 2379 (2002) p. 328-347
2. Favre J.M., "Towards a Basic Theory to Model Driven Engineering", 3rd Workshop in Software Model Engineering, WiSME 2004 (2004)
3. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and Dynamic Software Composition with Sharing. Seventh International Workshop on Component-Oriented Programming (WCOP02), , Malaga, Spain (2002)
4. Peltz, D.: Web services orchestration. A review of emerging technologies, tools and standards. Hewlett Packard, Co. January (2003)
5. van Ommering, R., Beyond Product Families: Building a Product Population ?, F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951 (2000) pp. 187-198
6. Estublier, J., Vega, G., Ionita, A.D.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications, Lecture Notes in Computer Science. Proceeding of MoDELS/UML Conference, Jamaica 3713, (2005) 69 – 83.
7. Estublier, J., Ionita, A.D., Vega, G., Relationships for Domain Reuse and Composition, Journal of Research and Practice in Information Technology, 38, 4, (2006) 287-301
8. Riebisch M., Streitferdt D., Pashov I. – Modeling Variability for Object-Oriented Product Lines, Workshop Reader of 17th European Conference on Object Oriented Programming *ECOOP 2003.* – Darmstadt, Allemagne, (2003) p. 165-178