

Relationships for Domain Reuse and Composition

Jacky Estublier

LSR-IMAG
220, rue de la Chimie BP53 38041 Grenoble Cedex 9 France
jacky.estublier@imag.fr

Anca Daniela Ionita

LSR-IMAG
220, rue de la Chimie BP53 38041 Grenoble Cedex 9 France
anca.ionita@imag.fr

German Vega

LSR-IMAG
220, rue de la Chimie BP53 38041 Grenoble Cedex 9 France
german.vega@imag.fr

Domain engineering is a very efficient approach for narrow and stable domains. However, building wide-scope applications often involves crosscutting several domains, such that domain composition becomes a necessity. Our approach aims at reusing existing domains, without changing or adapting them, even when they have been independently designed and implemented. The solution consists of establishing relationships between formal and executable domain models. Domain composition requires non-standard relationships between concepts from different domains and a specific methodology for defining and implementing them. The paper shows the issues addressed and the solutions found in our Mélusine environment, based on the experience accumulated in developing composite domains for various industrial applications.

Keywords: Model Composition, Reuse, Domain Engineering, MDE

ACM Classification: D2.11 Software Architecture, D2.12 Interoperability, D2.13 Reusable Software

1. INTRODUCTION

Domain engineering covers software applications where one of the primary concerns is “systematic software reuse” (Simos, 1997) in a given domain of expertise. A domain can be seen as a set of systems built from common assets. It is usually based on a common architecture, implemented once and shared by all the domain applications, such that a new application only needs to implement its specificities. This approach is interesting if the common part is heavily reused and represents a large fraction of the complete application, but this characteristic is true only in narrow domains.

The main drawback of domain engineering is that it is successful only when limited to very specific domains, while most software applications span several different domains. The paper presents our solution, which consists of composing existing domains and reusing both of their concepts and implementation.

There are different approaches for domain engineering. In generative programming, also called Feature Oriented Programming, a generator takes as input a feature model and generates the executable code (Czarneski et al., 2000). In the Domain Specific Language (DSL) approach, a language is defined in which a given application is “programmed” and then compiled into executable code (Wile, 2001). Both approaches rely on language and compiler technology. Composing domains would require you to extend or to compose languages (Bravenboer et al., 2004); therefore, to build new compilers, which is too expensive and explains why these approaches do not propose any way to build composite domains.

Our work addresses domain composition from a model/metamodel point of view, instead of a language/compiler one. We show that, if domains are formally defined by a metamodel, domain composition can be simply performed through the definition of relationships among metamodels. The paper explores the concepts

and issues raised by this new approach and compares them with the current technology around UML. Our architecture, called software federation (Le-Anh et al., 2003), is based on two main tenets:

- rely on a *metamodel*, gathering the characteristics that are common to all the applications in a domain;
- compose domains by defining *relationships* between domains at metamodels, models and instances level.

The paper presents the principles of the approach and the methodology adopted in the Mélusine environment for composing domains. We show how the semantics that need to be added when composing two domains can be supported by inter-domain relationships *only*. Doing so, the additional behaviour of the composite domain is fully external to the composed domains, allowing complete reuse, both conceptual (the metamodels and models) and technical (the implementation and tools). Executing a composite domain involves the transparent execution of the composed domains, as if they were used in isolation, thus allowing a high-reuse capability, which is a distinctive feature in our approach.

The approach requires the clear identification and separation of (1) the elements common to all domain compositions, (2) those common to all applications and (3) those specific to a particular application. The paper shows that this separation involves a specific methodology and shows which are the issues to be solved. The paper also provides the initial rules and guidelines, extracted from our experience in developing federations that help designers in solving such issues as making executable models, synchronizing domain states, managing lifecycle relationships and enforcing persistency.

The paper is structured as follows. Section 2 defines domain modeling; section 3 presents the principles of domain composition. Section 4 presents the method we use to compose domains. Section 5 relates our approach to other works on relationships and section 6 concludes the paper.

2. COMPOSING DOMAIN MODELS

Domains can be composed at different levels, and using different techniques. Our approach emphasizes a composition at meta model level, using relationships between concepts pertaining to these meta models.

2.1 DOMAIN MODELS AND META-MODELS

A domain can be defined from different points of view:

- ontology - classifying the domain elements;
- variations - identifying the differences among the applications that can be realized in the domain;
- domain modelling - defining the set of applications that can be realized in the domain.

The first point of view leads to the definition of classes of entities that, together, constitute the domain (Luoma et al., 2004). The second point of view leads to a type of Domain Specific Language (DSL) in which only variations (features) can be expressed (Falbo et al., 2002). The last one leads to the definition of a domain model, which is a mixture of the other ones that borrows its concepts and terminology from Model Driven Engineering (MDE) (Kent, 2002). Our approach falls into this last category.

A domain model :

- defines the concepts (class of elements) relevant from the application point of view, along with their relationships;
- is formalized as a meta-model (i.e., the definition of a language) in which the domain applications can be modelled (programmed).

The domain model defines a language (a Domain Specific Language) where the fundamental, high-level concepts and their semantics are first class objects and where (some of) the domain variations can be expressed (Estublier et al., 2005b).

This approach clearly separates the common part, shared by all applications in the domain, which is represented by the primitive language constructs, from the specific aspects of each individual application, which are defined in a model conform-to the metamodel (a program in the domain language). As often in UML, we transform the concepts identified in the metamodel into (Java) classes, operations into methods and the common behaviour into Java code. The models defined later by the application designer are transformed into instances of the (Java) metamodel classes. Our framework allows models to be mapped to various tools, which may be either

developed by us or available on the market; they might have been developed before the creation of the domain. The operational perspective (the tools) and the descriptive one (the model) are independently supported.

It is important to emphasize that the development of the metamodel and its behaviour consumes most of the effort, but it is done once for all. Then, the applications pertaining to the domain are only defined by a model executed according to the metamodels (Estublier et al., 2005c). If the model of the application only introduces structural variations (as is often the case), then defining an application in the domain needs no additional programming, but only a simple design consisting of defining the model and, optionally, selecting the desired tools for implementing it (Figure 1 :).

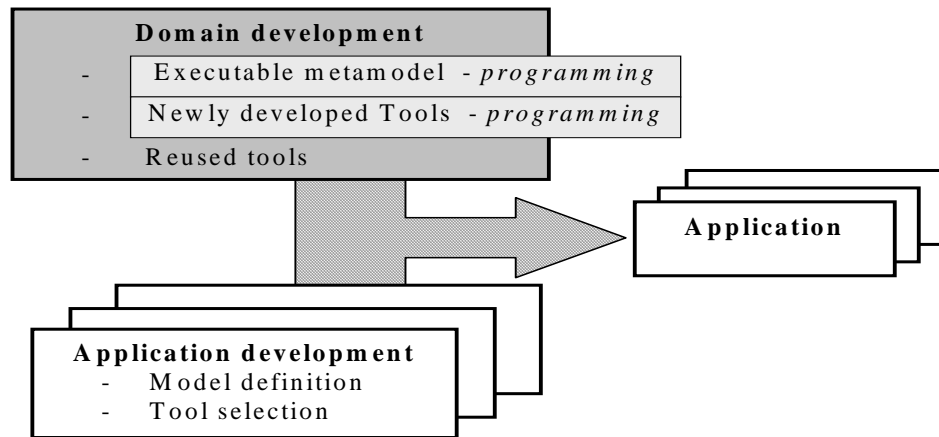


Figure 1 : Reuse of domain development for application development

2.2 RELATIONSHIPS FOR DOMAIN COMPOSITION

If a domain is seen as a set of systems to be built from common assets (in a given specific technical or business area), the basic rule for creating a domain is to define a metamodel that only contains the high-level, common concepts required by this set of systems and to let the variability among these systems to be defined into models and tools. The advantage is that, at the metamodel level, a domain is easy to understand and manipulate; the drawback is that, being very simple, the metamodel may not support some concepts required by unplanned future applications. As, by nature, a domain is narrow in scope, a typical application will span more than one domain. If domains are to be reused, there is a need to *compose* domains that may have been independently defined and developed.

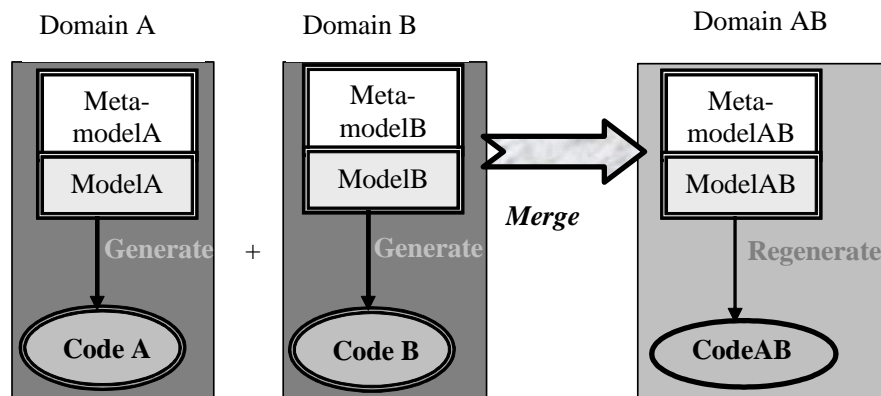


Figure 2 : Composition by model merging and code generation

Generally, composition is treated as a merge, followed by code regeneration; models A and B have usually the same metamodels (e.g. UML), such that the problem is “just” merging the models. In model merging, the original code for A and B and the original models developed in A and B domains cannot be reused; new code and new models have to be produced. The approach is reasonable only if the code can be fully generated, as in generative programming, and if the models are not too complex. In all the other cases, the approach does not facilitate the reuse of models and tools, which is essential in domain engineering.

In contrast, our system heavily relies on both metamodel and model composition. UML metamodel extensions are proposed by Clarke (2002), while a Hyper/UML approach is given by Philippow et al. (2003). The problem of metamodel composition is seldom addressed in literature, an exception being the GME environment presented by Ledeczki et al. (1999), which allows DSL composition, but not code generation

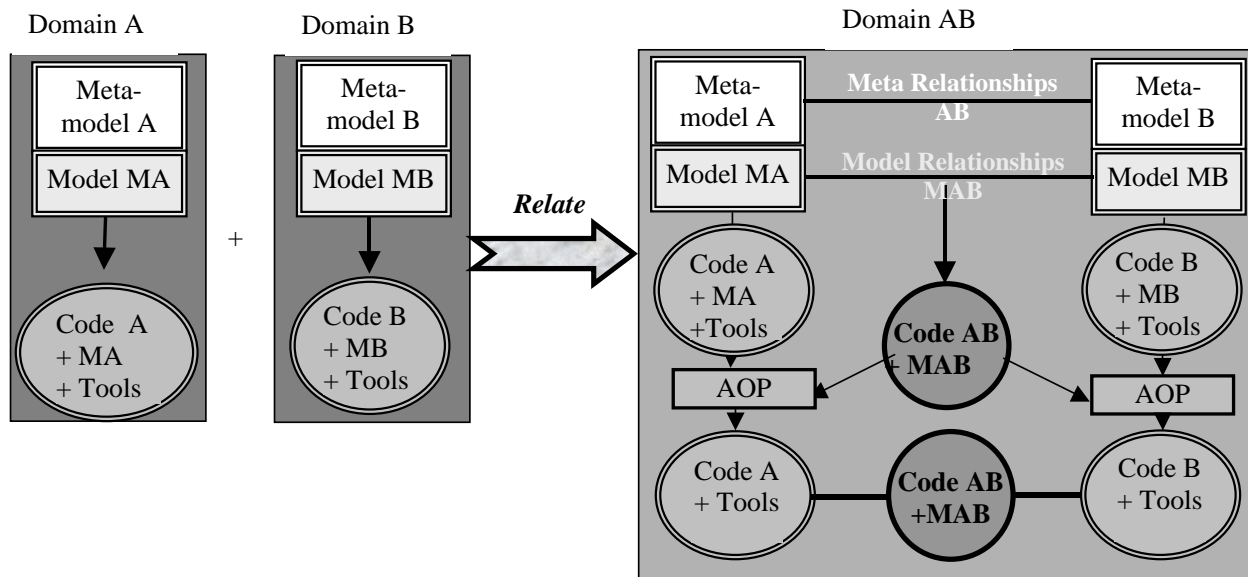


Figure 3 : Composition by relating models and full reuse of domain code

In order to reuse code and models as they are, instead of merging them (that may lead to a fairly different, new model and to a large amount of new code to be written) we limit ourselves to establishing relationships among the existing metamodels and among their models (Figure 3 :). Since we decided not to change the original domains, these relationships must contain all the semantics added when composing the domains. Thus, reuse does not only concern the conceptual part of the domain, but its implementation also, including COTS (commercial off-the-shelf) components and legacy tools, that may be very complex and have already been tested in various practical situations.

The reuse constraint requires that no changes are made to the metamodel, nor the code obtained after its (partially manual) transformation in Java. In particular, this means that the relationships and their semantics needed for domain composition cannot be transformed into new code added to the original Java source code. Nevertheless, collaboration requires navigation among domains. For that reason, the relationship semantics is transformed into “aspects” in the AOP (Aspect Oriented Programming) sense; to do so, an AOP machine is used for capturing method calls and modifying the byte code.

3. DOMAIN COMPOSITION METHODOLOGY

Composing domains, as proposed above, is an unusual task. It required several years and many realizations before identifying rules of thumb that help in the composition process. These rules and lessons are given here, but the methodology still has to be refined.

A domain has three layers:

- the domain **metamodel**, characterizing what is common for all the applications inside that domain;
- the application **model**, conforming to the metamodel, giving the specificities of an application in the domain;
- the **instances** generated when executing the application.

This architecture follows a reuse principle similar to those of product lines (Bosch, 2000): the invariant concepts of the domain are gathered in the metamodel and shared by all the applications, such that an application is only defined through a model that indicates its variabilities or specificities with respect to this common part. In product lines, as well as in our case, to get a good degree of reuse, domains are rather narrow; two or more domains have to be composed in order to develop large-scale applications.

Given this three-level architecture, domain composition through the definition of relationships has to also be performed at these three levels (see Figure 4 :). However, the issues and concepts are fairly different at each level, as explained in the following sections, starting from an example that represents a simplified version of one of our composite domains, which has already been reused several times.

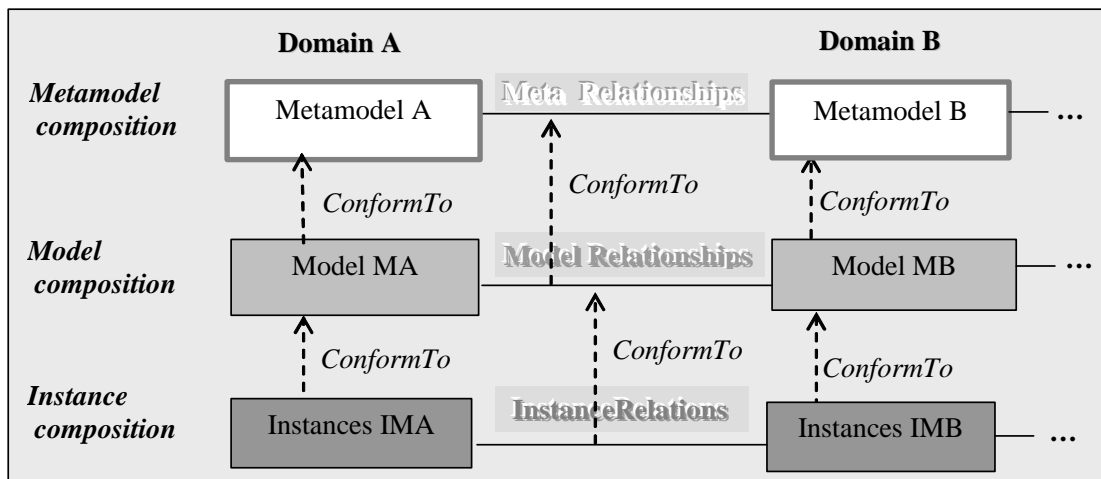


Figure 4 : Relationships at federation levels

3.1 Metamodel composition

Case study - Apel composition Since our original expertise was around configuration management and workflow support, it is not surprising that our first composition, called Apel, was between a “pure” *Workflow domain* and “pure” *Product domain*. This composition proved to be very valuable and has been reused in a number of industrial applications, including content management (Actoll) and memory design management (ST Microelectronics).

The metamodel level, illustrated in Figure 5 :, contains two examples of composition: *Workflow – Resource* and *Workflow - Product*. The *Product domain* was designed to manage a repository of typed artifacts that are versioned using branches and revisions. The *Workflow domain* allows for the definition of working models, composed of typed activities that consume and produce data through ports. The *Resource domain* registers the employees, their roles and other characteristics.

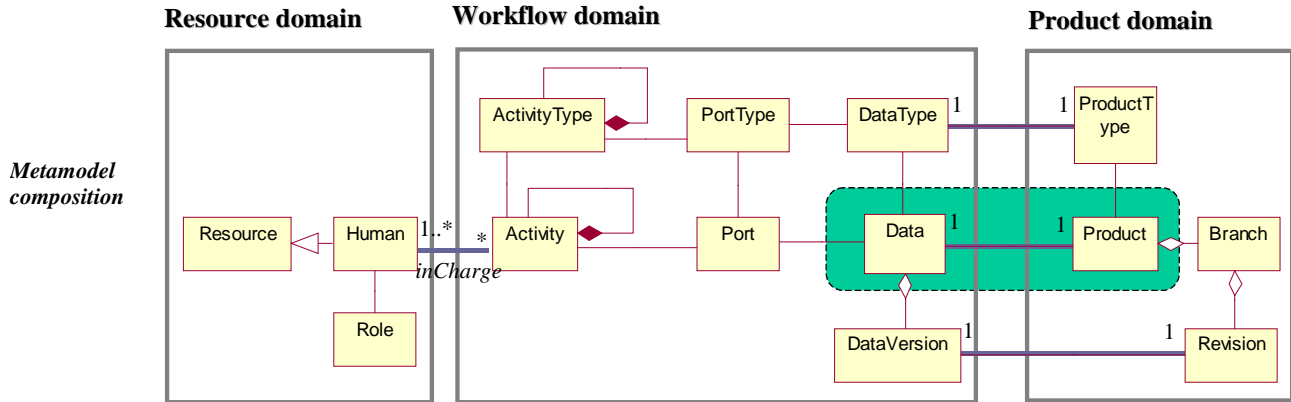


Figure 5 : Apel composite domain

This section deals with metamodel composition only. The composition between *Workflow* and *Resource* metamodels is done with the **horizontal relationship** between *Human* and *Activity* (see Figure 5 :); this is needed because a person should be in charge of an activity, that person should be an employee and therefore registered in the *Resource* domain. The composition between the *Workflow* and *Product* metamodels involves three **horizontal relationships**: *DataType* – *ProductType*, *Data* – *Product* and *DataVersion* – *Revision*. These relationships aim at providing transparent persistent capabilities to the data managed by the workflow. The relationship *Data* – *Product* is further explained from the point of view of model and instance levels in the sections 3.2.1 and 3.3.1.

3.1.1 Metamodel-level Relationships

Two types of relationships can be established among concepts from different domain metamodels:

- *Correspondences*, which relate the same or overlapping concepts.
- *Associations*, for navigation between concepts that have to collaborate.

Correspondences

As domain models are designed independently, they often define similar concepts in different ways (i.e., views), since each one outlines the characteristics important for its own purpose. For example, the *Data* and *Product* concepts from Figure 5 : have similarities, but, since they are designed for different purposes, they contain different information and propose different operations. Nevertheless, one can consider that they are two points of view of a unique abstract concept that subsumes both of them; for that reason, a data is always associated with a product and vice versa. This relationship is called a correspondence. The tuple $PersistentData = \{Data, \text{correspondence}, Product\}$ implements the de facto abstract concept “*PersistentData*”. More generally, a correspondence between two concepts means the identification of a new concept implemented as a tuple $CAB = \{CA, \text{correspondence}, CB\}$ that subsumes the original CA and CB concepts.

Managing a concept in this way raises a number of issues. Indeed, each original concept is still a part of an independent domain, which uses it in a certain way. A first difficulty is to identify, when an operation is performed on CA, which are the consequences on CB or, more generally, what is the correspondence between a state change of CA in a domain and the state of CB in another domain. For example, operation *editAttributes* applied to *Data* in the *Workflow domain* should be associated with operation *setAttributes* applied to *Product* in the *Product domain*, because the corresponding product has supplementary attributes, which the user of an application pertaining to the composite domain may wish to modify.

The second difficulty is that the new concept CAB may also have additional attributes and behaviour that are not in CA or CB and CAB may have also an operation *editAttribute*. The solution is to model the correspondence as an association class that supports the additional information and the semantics of the operations related to the

correspondence for synchronizing the states of the composed domains. UML extensions for modelling these relationships were given in another paper (Estublier et al., 2005a).

For establishing the correspondences, our experience with federations led to the identification of the following rules of thumb:

Rule 1: If two concepts are related by a correspondence and they aggregate other concepts, expect correspondences between some of their parts to exist too. For example, as *Data* aggregates *DataVersion*, it is natural that, if there is correspondence between *Data* and *Product*, it is likely that there is a correspondence between their parts, notably *DataVersion* and *Revision*.

Rule 2: It is useful to classify operations to easily identify the correspondences between elements of the same categories. For example, the operations may be categorized as: constructors, accessors, relationship related methods, viewers, persistency managers. A concept operation is likely to be related to an operation from the same category in the corresponding concept (e.g., a constructor of *DataVersion* to a constructor of *Revision*, because each time a *data version* is created, a *revision* has to be created).

Associations

Composing two domains means defining collaboration between these domains, which requires navigation between concepts pertaining to each one of them. For example, the association *inCharge* is found between *Activity* and *Human* in Figure 5 ; this association expresses the fact that a person is in charge of an activity. Associations express the fact that the two concepts are semantically related, which means that actions on one of them may have side effects on the other one. For example, when an activity starts, a notification should be automatically sent to the human in charge of that activity. Traditionally, this behaviour is programmed inside the activity concept. Since reuse requires that no changes are made to the original concepts, associations are also modelled as association classes supporting the additional behaviour.

These meta-relationships and their behaviour are defined once for a given domain composition. Altogether, the meta-relationships and the composed domain metamodels constitute the metamodel of a new composite domain and, as such, are shared by all applications pertaining to that composite domain.

3.2 Model composition.

3.2.1 Case study - Meta-level relationship instantiation

Consider the correspondence between *Data* and *Product* only. At the model level, there are more than one (model) elements conforming to (e.g. instance of) the same meta concept. For instance, in the *Workflow* domain, both *Document* and *ExperimentalData* are (conform to) *Data*, while in the *Product* domain, *Specification* and *Tool* are (conform to) *Product*. Provided two models *PM* and *WM*, pertaining respectively to the *Product* domain and *Workflow* domain, the problem is to be able to instantiate the meta-relationship *Data / Product* between elements of *PM* and *WM*, i.e., to determine which element of *PM* should be connected to which element of *WM*. In our example, *Document* is associated with *Specification*. General rules for instantiating meta-relationships are provided in the next section.

3.2.2 Relationships

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 for each domain and then define the relationships between these models.

The conformance between a metamodel and a model can be implemented in different ways. For the sake of clarity, let us suppose that this relationship is “instance_of” between a metaclass and a class. It means that the (meta) relationships defined at the meta level, between two metaclasses (like *Data* and *Product*), must be instantiated between the classes (i.e., instances of the metaclasses) found in the models. Given a relationship between two concepts at the meta-level, one should determine the relationships involved between their corresponding model entities. More precisely, given the meta-relationship {CA, R, CB}, CAE the set of instances of CA (the CA extension), CBE the CB extension, one should instantiate relationships (ca-r-cb), with $ca \in CAE$,

$cb \in CBE, r \in R$. There is no general solution for this; it depends on the R semantics. Nevertheless, the federation supports two types of solutions:

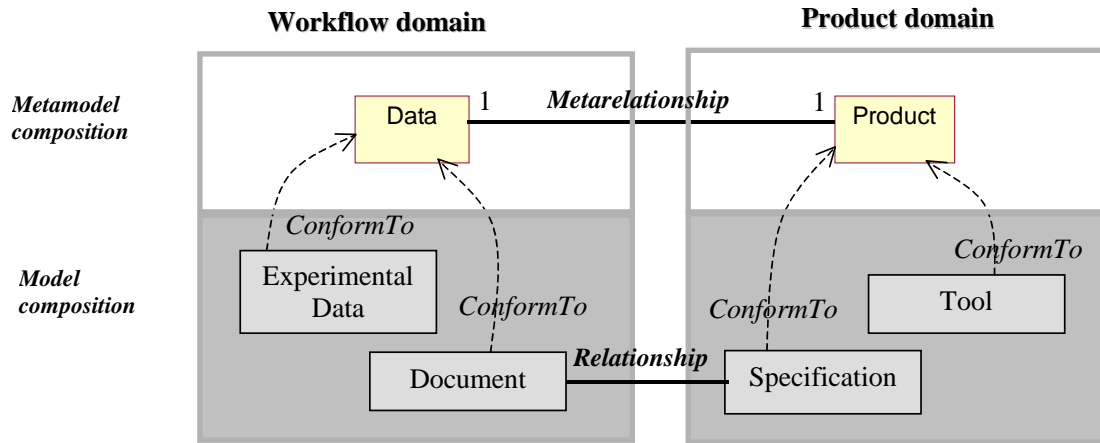


Figure 6 : Instantiation of meta-relationships and relationships

- *automatic instantiation.* The model composition designer provides a mapping function $f_r(ca) = \{cb\}$, which returns a sub-set of CBE, if provided $ca \in CAE$; conversely, mapping $f_r^{-1}(cb)$ might also be necessary. Taking the example from Figure 5 :, for every class conforming to *DataVersion*, a corresponding class conforming to *Revision* may be created in the other domain, having the same name (or a name that can be computed from the *DataVersion* entity). If the models have been created independently, a usual mapping technique is to define key attributes in CA and CB and to associate them according to the key values. By default, the “Name” attribute is supposed to be a key.
- *manual instantiation.* For each $ca \in CAE$, the model composition designer has to select explicitly a $cb \in CBE$ element, as in the example from Figure 6 :, where a user interface is needed to decide that *Document* corresponds to *Specification*.

At the model level, the relationship “r” relates two classes that have a specific behaviour and, therefore, “r” may involve a specific behaviour too, in order to coordinate the execution of the specific operations defined in these classes. In this case, for each relationship (ca, r, cb) , “r” holds specific semantics and “r” is a relationship class of its own. Experience shows that, in many cases, as in our example, models are purely structural and consequently “r” has no specific behaviour.

Returning now to the tasks of an application designer, if the domain is composite, he or she must:

- select the models to compose;
- define the mapping between model elements;
- define the specific semantics of each mapping.

Indeed, “programming” an application is limited in our case to performing the three activities above. Only the third one involves programming, but since the models do not often introduce behavioural variations, developing an application may not require any programming at all (Figure 7 :).

It is important to emphasize that model composition is to be performed only once for each model pair, i.e., for each application in the domain. For that reason, the manual selection and specific relationship semantics are often tractable. Nevertheless, automatic instantiation is appreciated, since it relieves the model designer from that burden; it is particularly required when models are very large, for example when CA and CB are two databases.

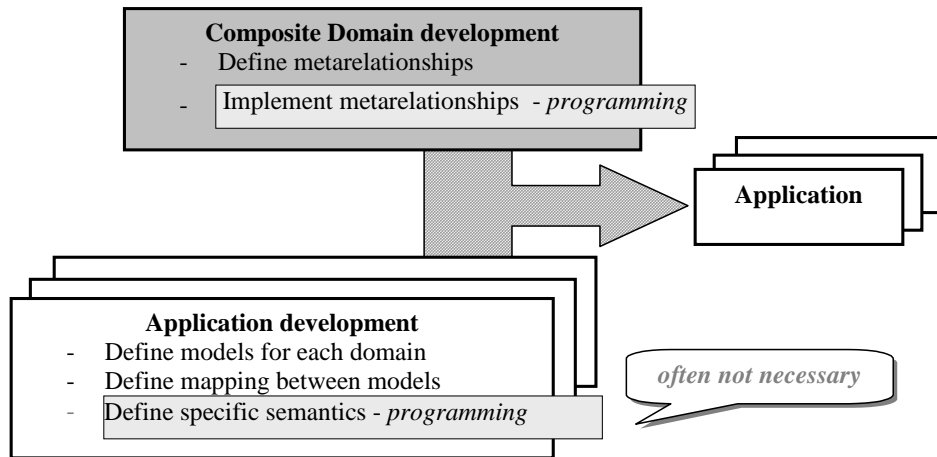


Figure 7 : Application development for a composite domain

3.3 Instance composition

3.3.1 Case study - Relationship instantiation

Let us now follow our example from Figure 6 : at the instance level. The model element called *Document* from the *Workflow domain* has multiple instances: *DocJacky*, *DocGerman*, *DocAnca* (see Figure 8 :). On the other side, *Specification* in *Product domain* has instances *MélusineSpec*, *ApelSpec* and *ProductDomainSpec*. As the relationship at the model level is one-to-one, one should determine which instance of *Specification* is linked to which instance of *Document*. In this case, the link is not performed automatically, but through manual selection. Some general considerations on establishing links are given in the next section.

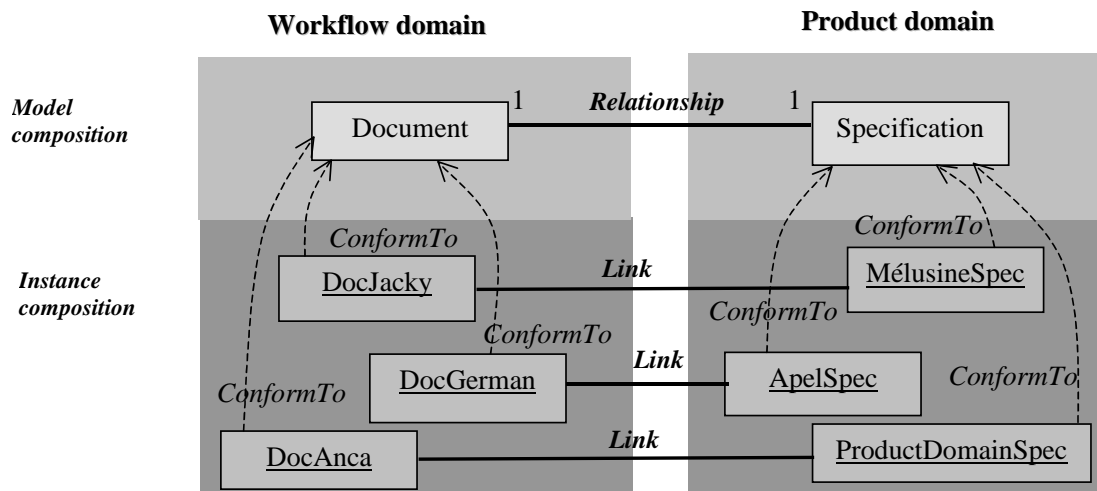


Figure 8 : Links for the Document – Specification relationship

3.3.2 Links

We have made the hypothesis that model elements are classes; therefore, at execution, models are “instantiated” in order to be executed. This instantiation is the classic object-oriented class/instance relationship.

Each class may have many different instances; therefore a mapping has to be provided also. This looks similar to instantiating relationships at the metamodel level, but at the model level the instances are statically known,

while at the instance level, new instances can be created at any time during execution; therefore, new relationships are to be instantiated dynamically. In the general case, it is not possible to know a priori which instances will be created.

Similar to meta-relationship instantiation, there are two solutions:

- if a mapping function is provided, the instantiation is automatic (e.g., in the example from Figure 8 :, an alternative solution might have been to match documents and specifications by their names);
- if a mapping function cannot be defined, the end-user is invited to manually select which element(s) of CB are to be linked.

Note that the mapping function either returns an existing element or creates an element. For example, creating an activity involves selecting an existing user; there is no automatic mapping and it is the team leader who selects the person to do the job. However, creating a certain document (e.g., *Doc_Jacky*) may require you to create a new Specification instance (e.g., *Spec_Jacky*).

The above discussion has fundamental consequences:

- Instances and links do not have specific behaviour: no programming is required for linking instances;
- Model elements must be reified at execution.

The first point means that the user does not have to define any additional behaviour. Most often the user has nothing to do (if the mapping is automatic), but sometimes he/she has to specify the corresponding instances; this user interaction is required, for example, to select which human will be in charge of a newly created activity.

The second point is a consequence of the fact that, at execution, links are to be instantiated between these model elements (Java instances). This defeats the classic approach, where models are compiled and executable code is generated. In practice this means that models must be executable.

3.4 Implementation issues

The sections above present a purist view; the reality is somehow different. Some simplifications are possible and, indeed, the purist approach presented above is seldom used.

3.4.1 Metaclasses, classes and instances

So far we have made the hypothesis that metamodels contain metaclasses and models contain classes, which can express new abstractions and specific behavior. Fortunately, this is often not the case because the domain engineering approach seeks to push most, if not all, of the domain semantics to the metamodel level.

It has been found in a study (Tibault, 1998) that only 15% of the studied domain metamodels provided user-defined types and only one third of them provided user-defined functions. Experience shows that most DSLs (Domain Specific Languages) do not provide any way to define new abstractions and, therefore, models do not require any programming at all. This is very important for two reasons.

From a human perspective, the application designer defines a new application without any programming (only by composing the models). It is one of the main reasons for following a domain engineering approach.

From a technical perspective, since Java does not provide metaclasses, a metaclass/class implementation is not direct and requires some workaround. In our V3 implementation, the “classic” solution was adopted and classes were generated to represent (reify) the model elements. Currently, instead of metaclasses, the metamodel contains “class type” concepts, i.e., classes whose instances represent types. The model elements are not specific classes, but “instances” of these “class types”; at execution, instances are ontologically related to the corresponding type instances. Doing so, models contain (Java) objects only and the relationship between a model and its instances is based on copies and clones. Distinguishing types also provides clues for the domain designer on how to define correspondences. Therefore we introduce two new rules of thumb:

Rule 3: If a domain concept represents a type, it should only be related to a type from the other domain. For example, *DataType* is related to *ProductType* in Figure 5 :.

Rule 4: If two types are related, one should seek correspondences between their “instances”. This means in our example that, if a *data type* corresponds to a *product type*, all the *data* having this type should be related to *products* that have the related *product type*.

3.4.2 Implementing relationships

In Java, relationships are often implemented as class attributes, but since we are not “allowed” to change the original classes, these attributes are not added in the source code, but instead are inserted into the class byte code using an AOP machine. The relationship semantics are also represented as a specific association class.

Each relationship is designed as an association class designed according to the type of the meta-relationship to which it conforms (association, correspondence), giving support for the interactions they allow and for managing the lifecycle link. The association class is then implemented by an aspect, inheriting a predefined relationship class (aspect class); in particular, lifecycle methods `newLink`, `deleteLink` and `navigateLink` are always available. These methods, and those specific to the relationship semantics, are called when operations are executed on the entities. For example, after the constructor is called on an object, the AOP machine calls the `newLink` method of the association class for creating a link and it either calls the mapping function, if defined, or it prompts the user.

If the composed domains are persistent, then the relationships among them should also be persistent, such that they should be saved and recovered when necessary. For that purpose, `saveLink` and `recoverLink` methods are included in the “persistent” super class of the relationship. A summary of methods used for Mélusine relationships among domains is given below.

Relationship methods	Link Methods	
	LifeCycle	Persistency
<code>createRelationship</code>	<code>newLink</code>	<code>saveLink</code>
<code>deleteRelationship</code>	<code>deleteLink</code>	<code>recoverLink</code>
	<code>navigateLink</code>	

Table 1. Relationship methods in Mélusine

4. RELATED WORKS ON RELATIONSHIPS

Our collaboration with industry led us to the realization of several federations, in which 3 to 7 domains have been composed. These federations are currently operational. This experience revealed several issues that have also been encountered by others regarding the way structure and behaviour influence each other and how relationships are transformed from analysis to design and implementation.

Based on the law of Demeter, “friends” who know each other do not need to be explicitly connected. Our approach actually conforms to the new law of Demeter for concerns (Lieberherr, 2004): objects from different domains (concerns) are not “friends” and need structural support (an association) for interacting. Similar approaches based on designing the interactions with aspects are given by Blay et al. (2004) or by Lieberher, with aspectual collaborations (2003) and the mechanism for establishing transversal associations (2001).

Relationship semantics are very important for implementation and reverse engineering. Harrison et al. (2000) separate the semantics of associations from their implementation for mapping them to Java. Guéhéneuc (2004) identified the differences between aggregation and composition (related to managing the life cycle of their parts) in order to transform them and to recognize them in code. For similar reasons, Gogolla (1998) gave a decomposition of qualified associations, aggregations and compositions into simple associations, enriched with OCL (Object Constraint Language) constraints, transforming complex UML (Unified Modeling Language) concepts into basic ones that are easier to transform into code. In the federation, the semantics for composing the domains is captured in the meta-relationships.

The support found in Mélusine for treating relationships at execution level, with the methods presented in section 4.4.2, is similar to many other action languages like the one introduced in Executable UML (Mellor et al., 2002), which has direct correspondences in Small (Shlaer-Mellor Action Language). Table 2 presents methods encountered in different executable approaches for treating relationships. Some code patterns are identified in (Génova et al., 2003) where associations are supported by assessor, mutator and auxiliary methods related to relationship state and definition in the code generation tool called JUMLA (Java code generator for Unified Modeling Language Associations). UML 1.5. gives an Action Semantics model that supports such action languages. From the examples given in Annex B of (OMG, 2003) one can notice that other languages, like ASL

(Action Specification Language) and AL (BridgePoint Action Language) that are also able to create a link, destroy a link and navigate an association, either to a single object or to multiple objects. These three methods are found in all action languages, including the one developed in M elusine environment.

JUMLA			
Assessors Methods	Mutator Methods	Methods for state	Methods for definition
test link existence get linked instances	remove add	isValid numberOfLinks	isBidirectional isMandatory isMultiple getMIN getMAX
Executable UML, Small, Tall			
Link Actions	Link object actions		
create link traverse link deleteLink	create link object action traverse link action unrelate action		

Table 2. Action languages

5. CONCLUSIONS

The domain composition approach presented here is based on rigorous domain modelling, which leads to a three-layer structure: metamodel, model, instances. This architecture is necessary for reusing already implemented domains without performing any changes to them. Thus, the composition is defined through the definition of relationships among concepts belonging to the composed domains and using an AOP machine to connect the related classes. This approach involves:

- introducing particular types of relationships, as correspondences, not found in UML;
- defining a methodology for relationship design, dedicated to domain composition;
- finding technological solutions for relationship instantiation at the three levels.

The practice has shown that many simplifications are possible, due to the fact that most of, and often all, the semantics of the composition is captured in the metamodel relationships. This has a strong influence on methods and tools used for federations design and implementation.

After several years of evolution, the M elusine environment has proved that composition by establishing relationships is a feasible and viable approach, satisfying a number of software engineering requirements: high-level design, reuse of meta-models, models and implementations, conceptual traceability at all levels.

Acknowledgements: We are grateful to the whole Adele team that designed and developed the M elusine federation environment, most notably J. Villalobos, T. Le. This work was partly supported by a Marie Curie Intra-European Fellowship.

REFERENCES

- BLAY-FORNARINO, M., CHARFI, A., EMSELLEM, D., PINNA-DERY, A-M. and RIVEILL, M. (2004): Software Interactions, *Journal of Object Technology*, **3**(10): 161-180.
- BOSCH, J. (2000): Design and use of Software Architectures, adopting and evolving a product-line approach, Addison-Wesley
- BRAVENBOER, M. and VISSER E. (2004): Concrete Syntax for Objects. Domain-Specific Language Embedding and Assimilation without Restrictions. Proc. Of *19th ACM SIGPLAN conference*, Vancouver, Canada.

- CLARKE S. (2002): Extending standard UML with model composition semantics, *Science of Computer Programming*. Elsevier Science: 71-100.
- CZARNECKI, K. and EISENECKER (2000): *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.
- ESTUBLIER, J. and IONITA, A.D. (2005a): Extending UML for Model Composition, *Australian Software Engineering Conference*, Brisbane, Australia, 31-38.
- ESTUBLIER, J., VEGA, G., (2005b): Reuse and Variability for Large Software Applications. *Proceeding of European Software Engineering Conference ESEC/FSE'05*, Lisbon, Portugal, 316-325.
- ESTUBLIER, J., VEGA, G. and IONITA, A.D. (2005c): Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications, *Lecture Notes in Computer Science. Proceeding of MoDELS/UML Conference*, Jamaica. 3713: 69 – 83.
- FALBO, R.A., GUIZZARDI, G. and DUARTE, K.C. (2002): An ontological approach to domain engineering, *Proc. of the 14th Int. Conf. on Software Eng. and Knowledge Eng.*, Ischia, Italy :351 – 358.
- GÉNOVA, G., DEL CASTILLO, C.R. and LLORENS, J. (2003): Mapping UML Associations into Java Code, *Journal of Object Technology*. 2(5): 135-162.
- GOGOLLA, M. and RICHTERS, M. (1998): Equivalence Rules for UML Class Diagrams, *UML'98: Beyond the Notation – International Workshop*
- GUÉHÉNEUC, Y-G. and ALBIN-AMIOT, H. (2004): Recovery Binary Class Relationships : Putting Icing on the UML Cake, *OOPSLA'04*. Vancouver, British Columbia, Canada.
- HARRISON, W., BARTON, Ch. and RAGHAVACHARI, M. (2000): Mapping UML Designs to Java, *OOPSLA'00*. Minneapolis, MN, USA: 178-188.
- KENT, S., (2002): Model Driven Engineering, *IFM 2002*, **2335**. LNCS Springer-Verlag.
- LE-ANH, T., VILLALOBOS J. and ESTUBLIER J. (2003) : Multi-level Composition for Software Federations, *Proceedings of the 6th European Joint Conferences on Theory and Practice of Software (ETAPS 2003). Workshop on Software Composition*.
- LEDECZI, A., MAROTI, M., KARSAI, G. and NORDSTROM, G. (1999): Metaprogramable Toolkit for Model-Integrated Computing, *Engineering of Computer Based Systems (ECBS)*, Nashville, TN : 311-317.
- LIEBERHERR, K. and WAND, M. (2001): *Navigating through Object Graphs Using Local Meta-Information*, Tech. Rep. NU-CCS-2001-05, Northeastern University.
- LIEBERHERR, K., LORENZ, D.H. and OVLINGER, J. (2003): Aspectual Collaborations: Combining Modules and Aspects, *The Computer Journal* **46**(5): 542-565
- LIEBERHERR, K.J. (2004): Controlling the Complexity of Software Designs, *Int. Conf. On Software Engineering*.
- LUOMA, J., KELLY, St. and TOLVANEN, J.-P. (2004): Defining Domain-Specific Modeling Languages: Collected Experiences, *Workshop on Domain-Specific Modeling (DSM'04)*, Computer Science and Information System Reports, Tech. Rep., TR-33, Univ. of Jyväskylä, Finland.
- MELLOR, St.J. and BALCER, M.J. (2002): *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley.
- OMG Unified Modeling Language Specification, Version 1.5, Object Management Group, March 2003.
- PHILIPPOW, I., RIEBISCH, M. and BOELLERT, K. (2003): The Hyper/UML Approach for Feature Based Software Design, the 4th AOSD Modeling With UML Workshop, San Francisco, CA.
- SIMOS, M. (1997): Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions, *STJA'97 Conference Proceedings*, Technische Universität Ilmenau, Thüringen: 126-132.
- TIBAULT., S. (1998): *Languages Dédiés : Conception, Implémentation et Application*. PhD thesis. , Rennes, France.
- WILE, D. S.. (2001): Supporting the DSL Spectrum, *Journal of Computing and Information Technology*, CIT 9 (4): 263-287.