e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Bi-dimensional Composition with Domain Specific Languages

Anca Daniela Ionita*, Jacky Estublier**, Thomas Leveque**, Tam Nguyen**
*University Politehnica of Bucharest, Automatic Control and Computers Faculty
**LIG-IMAG, Grenoble, France
Anca.Ionita@mag.pub.ro, Jacky.Estublier@imag.fr, Thomas.Leveque@imag.fr, Tam.Nguyen@imag.fr

Abstract

The paper presents how domain modeling may leverage the hierarchical composition, supporting two orthogonal mechanisms (vertical and horizontal) for composing completely autonomous parts. The vertical mechanism is in charge of coordinating heterogeneous components, tools or services at a high level of abstraction, by hiding the technical details. The result of such a composition is called "domain" and represents a high granularity unit of reuse, which may be easily developed in Mélusine framework. A domain is characterised by a Domain Specific Language (DSL) and applications in that domain are defined by models executed by the DSL interpreter. Most often, this is significantly simpler than writing a program using a general purpose language. Unfortunately, DSLs have a narrow scope, while real world applications usually span over many domains, raising the issue of domain (and DSL) composition. To overcome this problem, the horizontal mechanism composes domains at the level of their DSLs, even if they have been independently designed and implemented. The paper presents a model and metamodel perspective of the Mélusine bi-dimensional composition, assisted and automated with the Codèle tool, which allows specification at a high level of abstraction, followed by Java and AspectJ code generation.

1. Introduction

In the widely adopted Component Based Software Engineering (CBSE) approach, components know each other, must have compatible interfaces and must comply with the constraints of the same component model, which reduces the likelihood of reusing components, and therefore the capability to obtain a large variety of assemblies. Therefore, alternative composition mechanisms have to be explored, such as to preserve the CBSE advantages (coming from hiding the internal structure and reusing components without any change) but to relax the rigidity of the composition constraints:

– The components or, generally speaking, the parts, should ignore each other, such that they could have been designed and developed independently, i.e. they do not call each other;
– Composed parts should be of any nature (ad hoc, legacy, COTS, local or distant);
– Parts should be allowed to be heterogeneous i.e. they do not need to follow a particular model (component model, service etc.);
– Parts should be reused without having to perform any change in their code.

The bi-dimensional composition mechanism presented here is intended to be a solution for such situations. The idea is to obtain composable elements that are not traditional components, but much larger units, called domains, which do not expose simple interfaces, but domain models, representing DSLs for specifying the application models.

One of the important problems to be solved was related to the heterogeneity of components, tools or services that have to be reused. A possible solution was to imagine that the part to be composed is wrapped into a “composable element” [22]. There was also a need to define a composition mechanism that is not based on the traditional method call, for composing parts that ignore each other, and therefore do not call each other. The publish/subscribe mechanism [2] was an interesting candidate, since the component that sends events ignores who (if any) is interested in that event, but the receiver knows and must declare what it is interested in. If other events, in other topics, are sent, the receiver code has to be changed. Moreover, the approach works fine only if the sender is an active component. A more appropriate solution for our requirements could be given by Aspect Oriented Software Development (AOSD) [18], [13], which eliminates some of the constraints above, since the sender (the main program) ignores and does not call the receiver (the aspects). Unfortunately, the aspect knows the internals of the main program, which defeats the encapsulation principle [8] and aspects are defined at a low level of abstraction (the code) [12], [24].

In our approach, heterogeneity is dealt with by coordinating components, tools or services from a higher abstraction level; this is what we call *vertical composition* and is attained by defining a domain DSL, which can natively specify entities specific to the domain and natively grasp the semantics (behaviour) of these entities within its interpreter; therefore, defining an application in the domain turns out to be the simple definition of a model in the DSL language. As usual, each domain is well instrumented with editors, interpreters, debuggers, analyzers, whose development is rather expensive, even with the help of the recent environments. Maybe more important, the practitioners acquire expertise in using these languages and benefit from a large set of existing models, which constitute a part of the company assets. Therefore, a large scale reuse of these domains is essential for the applicability of such an approach and is promoted through rich DSL semantics. Unfortunately, the richer the semantics embedded in the DSL, the simpler the models, but the narrower the language scope. In this context, the main drawback of DSLs comes out from the fact that most real life applications usually crosscut several domains, but they cannot be simply described by selecting a set of independent domain specific models, each one describing how the application behaves inside each covered domain.

Consequently there is also a need to compose domains; this is what we call *horizontal composition* and is not based on calling component interfaces, but on composing domain DSLs and models. In contrast to method call, model composition does not impose that models stick to common interfaces, or know each other, because one can either merge or relate independent concepts. Moreover, model composition allows the definition of variability points [17], which makes the mechanism more flexible than component composition.

For building applications spanning different domains, the challenge is to reuse the domain tools, the existing models and the practitioner’s expertise and know-how; this is far from trivial and is not possible if one creates a new language for the composite domain. As discussed above, for obtaining a non-invasive method, a possibility is to adopt an implementation based on AOP (Aspect Oriented Programming); the composed domains and their models are totally unchanged and the new code is isolated with the help of aspects. However, since the AOP technique is at code level, performing domain composition has proved to be very difficult in practice; the conceptual complexity is increased, due to the necessity to deal with many technical details. This problem has been treated in many research works. The elevation of crosscutting modeling concerns to first-class constructs has been done in having [15], by generating weavers from domain specific descriptions, using ECL, an extension of OCL (Object Constraint Language). Another weaver constructed with domain modeling concepts is presented in [16], while [25] discusses mappings from a design-level language, Theme/UML, to an implementation-level language, AspectJ. Our solution is to clearly separate the specification of the composition from its implementation, by designing at a high conceptual level and then generating the code based on aspects.

For managing the complexity in a user friendly manner, the user defines the composition using wizards, for selecting among pre-defined properties. Designers and programmers are assisted by the Mélusine engineering environment for developing such autonomous domains, for composing them and for creating applications based on them [22]. For facilitating an easier domain composition, by generating Java and AspectJ code, Mélusine was leveraged by Codèle, a tool that guides the domain expert for performing the composition at the conceptual level, as opposed to the programming level.

Chapter 2 describes the architecture and the principles that stand behind the creation of domains driven by their DSLs and the composition at a high level of abstraction. Chapter 3 presents the metamodels that allow code generation for vertical and horizontal composition. Chapter 4 introduces some details related to the implementation choices, including some mappings for code generation. Chapter 5 compares the approach with other related works and evaluates its usefulness in respect with the domain compositions performed before the availability of the code generation facility offered by Codèle.

2. Bi-dimensional Composition Based on DSLs

The alternative composition idea presented above is to create units of reuse that are *autonomous* (eliminating dependencies on the context of use) and *composable* at an *abstract level* (eliminating dependencies on the implementation techniques and details). The solution presented here combines two techniques (see Fig. 1): *building autonomous domains* using *vertical composition* and *abstract composition of domains* using *horizontal composition*, performed between the abstract concepts of independent domains, without modifying their code.

2.1. Developing Autonomous Domains: Vertical Composition

Developing a domain can be performed following a top-down or a bottom-up approach. From a top down perspective, the required functionalities of the domain can be specified through a model, irrespective of its underlying technology. Then, one identifies the software artifacts (available or not) that will be used to implement the expected functionality and make them interoperate. From a bottom up perspective, the designer already knows the software artifacts that may be used for the implementation and will have to interoperate; therefore, the designer has to identify the abstract concepts shared by these software artifacts and how they are supposed to be consistently managed. Finally, one defines how to coordinate the software artifacts, based on the behavior of the shared concepts.

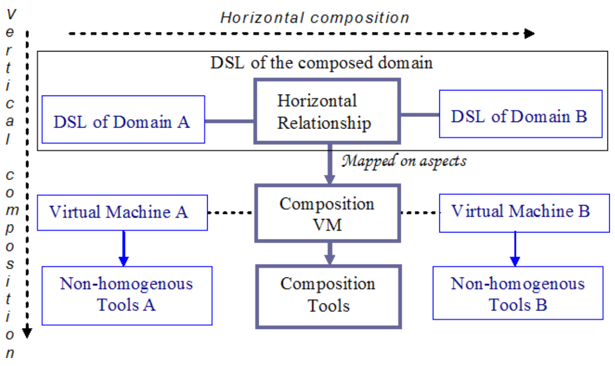


Figure 1. Bi-dimensional composition mechanism

In both cases, the composition is called vertical, because the real software components, services or tools are driven based on a high level model of the application. The model elements are instances of the shared concepts, which are abstractions of the actual software artifacts. The synchronization between these software artifacts and the model means that the evolution of the model is transformed into actions performed by the software artifacts.

The set of shared concepts and their consistency constraints constitute a domain model, to which the application model must conform to. In the Model Driven Engineering (MDE) vocabulary, the domain model is the metamodel, or a DSL for all the application models for that domain [12].

The application models are interpreted by a virtual machine, built according to the domain DSL, which orchestrates the lower level services or tools. The domain interpreter is realized by Java classes that reify the shared concepts of the domain model and whose methods implement the behavior of these concepts. In many cases, these methods are empty, because most, if not all the behavior is actually delegated to other software artifacts, with the help of aspect technology. Thus, the domain interpreter, also called the domain virtual machine, separates the abstract and conceptual part from the implementation, creating 3 layers architecture [12]. The domains may be autonomously executed, they do not have dependencies and they may be easily used for developing applications.

2.1.1. Domain Specific Languages in Mélusine

The domain specific languages defined in Mélusine are rather small, covering a narrow domain and typically, they are object oriented. As usual, each language description contains two parts: syntax and semantics. The abstract syntax (AS) of the language contains the concepts and rules necessary to define a valid model, while its Semantic Domain (SD) is needed to provide the meaning of the abstract syntax concepts. By convention, the abstract syntax is defined by a class diagram, while the Semantic Domain is defined based on the methods pertaining to the AS classes, plus some additional classes. The concrete syntax (CS) is provided by a specific editor.

The *Product* domain, one of our intensely reused domains, is presented in the case study of this paper. It was developed as a basic versioning system for various products, characterised by a map of attributes, according to their type; the versions are stored in a tree, consisting of branches and revisions. The *Product* domain DSL is shown in Fig. 2 and contains both AS elements (light colored) and SD elements (dark colored).

From a Language Engineering point of view, this DSL is the definition of a language in which models are written; from a Domain Modeling point of view, it is a model of the application domain [12]. Thus, the DSL is the symbiosis of both views, since it is a language in which models are written, but, being Domain Specific, it contains the domain specific concepts, their allowed relationships and their behaviors. The DSL captures both the abstract syntax and the semantic aspects and it has different purposes: on one hand, it is used to develop models; on the other hand it is used to develop the interpreter and the editor of the domain and to compose domains for enlarging their scope. These activities involve an awareness of the concepts related to the semantic domain, which is necessary, for instance, for developing the interpreters, but also for composing them, in order to be able to compose domains.

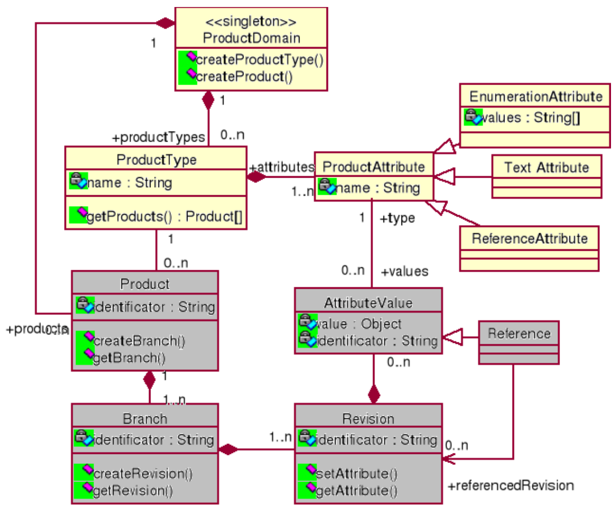
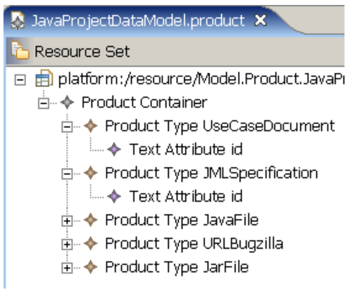


Figure 2. DSL of Product domain

2.1.2. Domain Specific Models

For defining an application, one creates a model that is going to be interpreted at run-time. Suppose we use our *Product* domain to version the software artefacts produced when developing an application based on the J2EE architecture. A *Servlet* in this application model conforms to the *ProductType* concept from the *Product* DSL.

In practice, the models can be expressed in several formalisms, and represented in a variety of ways; indeed, models may be defined in UML, or in Ecore, through generated editors (like in most metamodeling environments), and stored in different formats, currently XML based. We have developed a number of filters, allowing one to define models and metamodels in these different formalisms, using different environments and editors. An example of editor for *Product* domain is given in Fig. 3. However, since our DSLs are written in Java, models always consist in a set of Java objects at execution. Models, expressed in various formalisms, will be transparently converted to Java objects at the beginning of interpretation phase. In most cases, when domains are narrow enough, the complete models semantics lies in the DSL. In this case, models are purely structural, and simple editors like those generated by EMF are sufficient. This is very important, because it allows non programmers to define executable models themselves. If models require specific semantics, it has to be described in Java.



```
JavaProjectDataModel.product ×
Resource Set
⊟ platform:/resource/Model.Product.JavaPı
  ⊟ Product Container
    ⊟ Product Type UseCaseDocument
        Text Attribute id
    ⊟ Product Type JMLSpecification
        Text Attribute id
    ⊞ Product Type JavaFile
    ⊞ Product Type URLBugzilla
    ⊞ Product Type JarFile
```

Figure 3. Model editor for Product domain

2.2. Abstract Domain Composition: Horizontal Composition

It may happen that the development of a new application requires the cooperation of two concepts, pertaining to two different domains, and realized through two or more software components, services or tools. In this case, the interoperation is performed through a horizontal composition between these abstract concepts, and also through the domain virtual machines, ignoring the low level components, services and tools used for the implementation. The mechanism consists in establishing relationships between concepts of the two DSLs and implementing them using aspect technology, such as to keep the composed domains unchanged. A very strict definition of the horizontal relationship properties is necessary, such as to be able to generate most of the AOP code for implementing them. This code belongs to the Composition Virtual Machine and is separated from the virtual machines of the composed domains.

This composition is called horizontal, because it is performed between parts situated at the same level of abstraction. It can be seen as a grey box approach, taking into account that the only visible part of a domain is its DSL. It is a non-invasive composition technique, because the components and adapters are hidden and are reused as they are. The composition result is a new domain model and therefore, a new domain, with its virtual machine, so that the process may be iterated. As the domains are executable and the composition is performed imperatively, its result is immediately executable, even if situated at a high level of abstraction.

Model composition is actually performed by creating links between model elements (instances of the DSL classes) so by instantiating the horizontal relationships defined at metamodel level. The choices of links ends may be made either automatically or manually (interactive) with the help of the application designer. Interactive selection is often used, since concepts of existing models may not match to each other perfectly (they may have different names, but the same meaning or have the same name, but behaviors that partially overlap) and no rule can be defined for it. However, it may be a tedious process, especially for composing large models. In contrast, automatic selection can relieve model designer from this burden and is particularly appreciated when models are very large. The default criterion for automatic selection can be based on name matching.

2.2.1. Horizontal Composition at Metamodel, Model and Execution Levels

A real example of domain composition, realized in our industrial applications, is illustrated in Fig. 4. On the left, the *Activity* domain supports workflow execution, while on the right, the *Prod-*

uct domain is meant to store typed products and their attributes. Each domain has a DSL (see the metamodel level). The upper part shows the visible concepts (the abstract syntax, in light grey) used for defining the models with appropriate editors; the lower part (in dark grey) shows the hidden classes, introduced for implementing the interpreters (the virtual machines) and for holding the state of the models during the execution process.

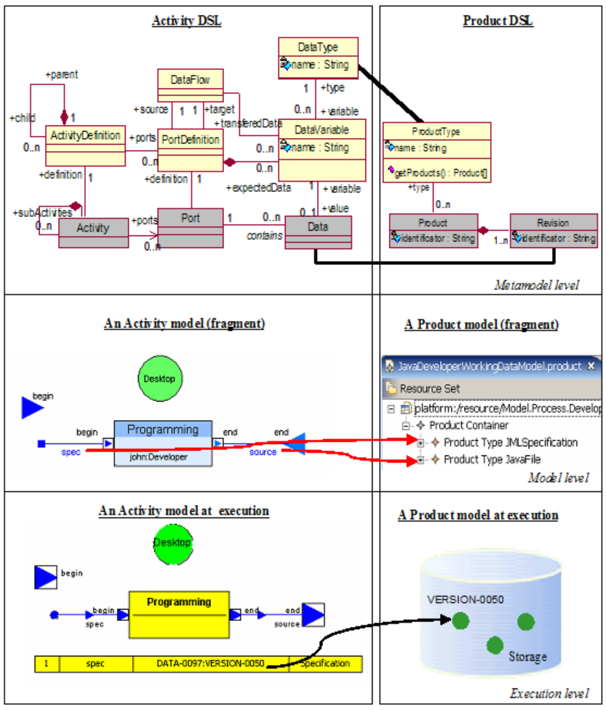


Figure 4. Composition of Activity and Product domains

For each domain, a model is made by instantiating the concepts found in the light colored part of the domain DSL. At model level, on the left, Figure 4 shows an Activity model, conforming to its DSL above. This model describes a very simple software development process, which only contains one activity – *Programming*; the box *Programming* is an instance of the *ActivityDefinition* concept. Labels on the activity connectors, like *spec* or *source* are instances of the *DataVariable* concept. These data variables correspond to instances of *DataType: Specification* and *Program* (not shown in this figure). Similarly, on the right side of Figure 4, at model level, there is a *Product* model, containing two instances of *ProductType: JMLSpecification* and *JavaFile*.

The *Activity* model in this example is made of a simple activity, in which a developer *john* receives a software specification *spec*, realizes the activity *Programming* and produces the source code *source*. However, the developer *john* may need to work on various revisions of his specification or of his source, so the *Activity* domain needs to be composed with the *Product* domain, for adding the versioning facility. These two domains (*Activity* and *Product*) are related together by horizontal relationships at metamodel level, for example, a horizontal relationship is defined between *DataType* and *ProductType* and another one between *Data* and *Revision*. At model level, a link relates the type of *spec* – *Specification* (found in the *Activity* model) – to *JMLSpecification,* instantiated from *ProductType* (found in the *Product* model). Another link relates *Program* (the type of *source*) to the *JavaFile* product type. These two links conform to the relationships defined between the *DataType* and *ProductType* concepts. At execution level, a data from the Activity model, for example DATA_0097 is related to a revision from Product model, for example *VERSION-0050* (see Fig. 4). This link conforms to the relationship between *Data* and *Revision*, situated at metamodel level.

Even if in the example above there was a clear correspondence between Specification from Activity model and JMLSpecification from Product Model, in practice, there may be several instances of a metamodel concept on both sides, as exemplified in Table 1. For creating the link at model level, one has to choose among these instances, such as to select a single one-to-one correspondence.

3. Metamodels for the Bi-dimensional Composition

3.1. Metamodel for the Vertical Composition

The methods defined in a *domain concept* are introduced for providing some *behavior* (see Fig. 5

Table 1. Different mappings of metamodel concepts on their instances at model level (for application development in Java and PHP respectively)

Domain	Product		Activity
Metamodel	ProductType		DataType
Model	(Java)	(PHP)	
	Use Case Document	Use Case Document	Requirement
	JML Specification	UML Specification	Specification
	Java File	PHP File	Program
	URL Bugzilla	Vision Project	BugReport

for the correspondent metamodel elements). In most cases, only a part (if any) of the *behavior* is implemented inside the method itself, because, most often, its functionality involves the execution of some tools. The notion of *Feature* has been defined to provide the code that contains one or more method *interceptions* and calls the services that actually implement the expected *behavior* of that methods. Additionally, a *feature* can implement a concern attached to that method, like security or persistency, which can be an optional *behavior*, as in product line approaches.

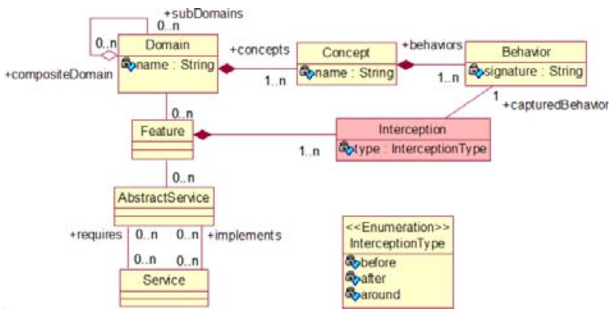


Figure 5. Metamodel for the vertical composition

For the vertical composition, the non-homogeneous units of reuse correspond to the generic notion of *Service* (see Fig. 5). At instantiation, they may correspond to components, tools, COTS etc. In our *Product* domain, the persistency *service* may be supplied either by SQL storage, or by a repository of another versioning system, like Subversion or CVS; the choice can be done by the client. For the example from Figure 4, the method *getProducts* of the class *ProductType* is empty and it is its associated *feature* that delegates the call to a database where actual products are stored. However, a *feature* is not related directly to *services*, but through *abstract services* – an abstraction for a set of functionalities defined in a Java interface, which are ultimately executed by components/tools representing the *services* (i.e. implementing its methods).

More than one *feature* can be attached to the same method and each *feature* can address a different concern. The word *feature* is used in the product line approach to express a possible variability that may be attached to a concept. Our approach is a combination of the product line intention with the AOP implementation.

Moreover, the purpose is to aid software engineers as much as possible, in the design and development of such kind of applications. By using the Codèle tool, which "knows" the metamodel from Figure 5, the software engineer simply creates instances of its concepts (*Behavior, Interception, Feature, Service* etc.) and the tool generates the corresponding code in the Eclipse framework. As well as all Mélusine DSLs, Codèle metamodels are implemented with Java, whereas AspectJ, its aspect-oriented extension, is used for delegating the implementation to different tools and/or components (instances of the *Service* concept).

3.2. Metamodel for the Horizontal Composition

In other similar approaches, as in model collaboration [26], AOP was mentioned as a possible solution for implementing collaboration templates in service oriented architectures (SOA), orchestration languages or coordination languages. As our approach is based on establishing relationships, it can also be compared to [1], where the properties of AOP concepts are identified (e.g. behavioral and structural cross-cutting advices, static and dynamic weaving). Our intention is to

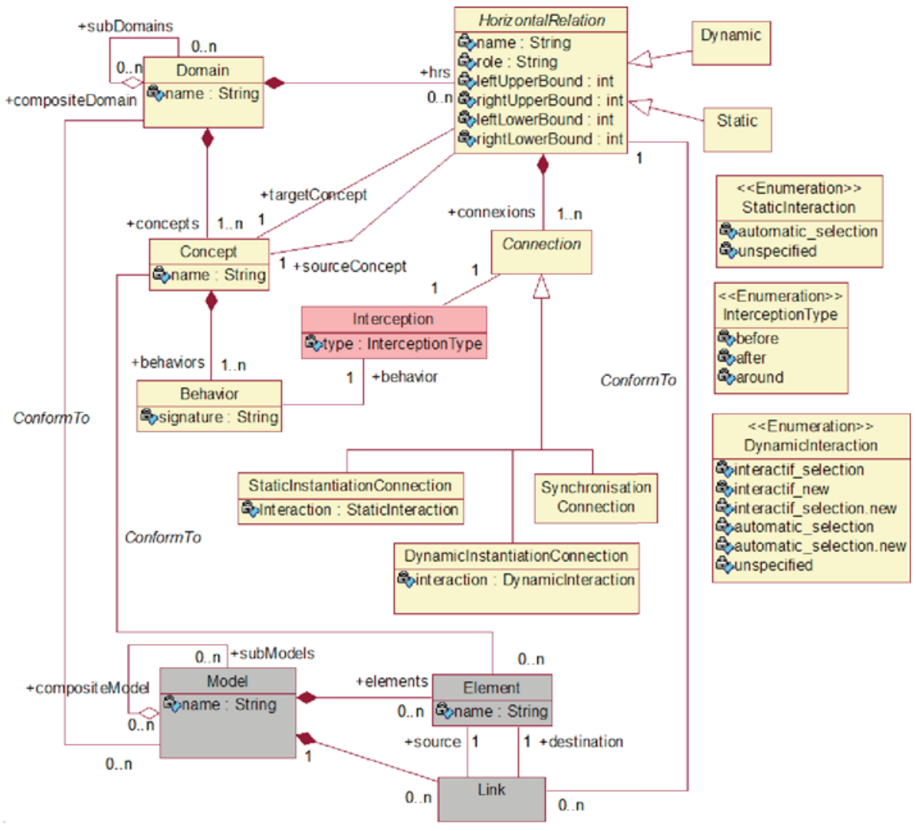


Figure 6. Metamodel for the horizontal composition

identify such properties at a more abstract level, such as aspects only constitute an implementation technique. The method we use for defining and generating horizontal compositions between domains is similar to transforming UML associations into Java [14], but using AOP, because we are not allowed to change the domain code.

To provide an effective support for domain composition, Mélusine requires a specific formal definition and semantics. The metamodel from Fig. 6 shows that domain composition relies on *Horizontal Relationship*, made of *connections*. A relationship not only represents a set of communication links between instances, but also expresses the interaction between them. The execution of an operation from an instance pertaining to one side of the relationship has consequences on the instances from the other side. In Codèle, such a piece of interaction is called *connection* and is established between a *source concept*, pertaining to the source *domain*, and a *destination concept* in the target *domain*. First, a connection performs the *interception* of the *behavior* (method) pertaining to the *source concept*, and then some computation, depending on its type: *Synchronization, StaticInstantiation, DynamicInstantiation*. Since *concepts* are reified as classes and operations are defined as their methods, a *connection* may be expressed based on the AOP mechanism: the method from one side is captured, allowing for the interaction with the methods from the other side. As a *concept* may have many methods, each one being able to participate to one or many *connection*(s), a *horizontal relationship* may manage many *connections*.

3.2.1. Composition Specific Semantics

From the experience gained while defining connections, some composition templates have been identified, such that some types of connections may be generalized and generated automatically. Connections are categorized according to their purpose:

– *Synchronization* – the most popular kind of connections, modifying the state of the instance at the destination end, with respect to the changes performed for the instance at the source end;

– *Instantiation* – in charge of creating an instance of the horizontal relationship (a link between elements of the models to be composed) and, eventually, also with the creation of the instance at the destination end.

For establishing a link between two instances participating in a horizontal relationship, two issues must be considered: 1) the moment of creating the link, and 2) the alternatives for setting the destination instance. These semantics are taken into account when instantiating HRs at model level (see Fig. 4).

1) *The moment of creating the link.* Most often, a link is established when creating the instance that must be the origin of the link. These instances (representing elements of the models to be composed) are created either before execution (if they conform to AS concepts and are part of a domain specific model defined for a domain to be composed) or during the execution (if they conform to concepts introduced for interpreting these models). Therefore, it is possible to establish links either before or during the execution; the two situations actually correspond to the two types of horizontal relationships: *static* and *dynamic* respectively. For doing so, the method for creating the source instance (e.g. the constructor) is captured by the AOP machine and extended with the creation or the reification of the link; for the links defined before execution (between elements of domain specific models of the domains to be composed) the link is reified when the model elements are reified, just before starting the execution. In our example from Fig. 4, the links created between models (i.e. before the execution) are called *static*, while the links created to relate these models at execution are called *dynamic*. For example, the link between *Specification* and *JML Specification* is static, whereas the link between *DATA_0097* and *VERSION-0050* is dynamic.

2) *The alternatives for setting the destination instance.* For deciding the link destination end, there are two kinds of mapping functions:

– *Creation* (returning a *new* instance) and
– *Selection* (returning an existing instance).

Either to create or to select a destination instance, one should define some criteria, often based on the properties of the source instance. For example, the mapping function may create a destination instance, providing the source name as parameter (creation mapping) or it may look for the destination instance with the same name as the source instance (selection mapping). Besides the two alternatives above, the mapping function may adopt two kinds of processes:

– *Automatic:* the destination element is found automatically, if the searching criterion is provided;
– *Interactive:* the destination element is found with human intervention, if the searching criterion is not provided.

For the *Automatic* case, by default, Codèle supports a searching criterion based on a key attribute, like *name* or *identifier*. The default criterion is used if no user-defined searching criterion is provided.

The combination of the mapping kinds and processes presented above gives diverse ways to set the destination instance and the dynamic interaction may follow several valid possibilities, as also presented in the metamodel from Fig. 6:

– *Automatic.New*: the mapping function automatically creates and returns a new instance;
– *Automatic.Selection*: the mapping function automatically returns an existing instance;
– *Automatic.Selection.New*: the mapping function automatically searches for an existing instance and, if not found, creates a new one;
– *Interactive.Selection*: the destination instance is selected by a human, and
– *Interactive.Selection.New*: first, a human tries to select an existent destination instance; if he or she does not find anything appropriate, it is possible to ask for the creation of a new one.

The above options may be valid or not. If a link is created at execution time, all the above options may be used for setting the link destination. However, if a link is created before execution, the only valid option is *Auto-*

matic.Selection, because the link already exists and it must be simply reified.

4. Implementation Issues

4.1. Implementation Choices

Our approach follows the language/interpreter technology. However, to be later composable with other domain interpreters, the DSL interpreter must follow conventions in the way the concepts defined in the metamodel are mapped to the target programming language.

First, the target implementation language must be able to express the DSL operational semantics. Since the metamodels are object-oriented, it is convenient to use an object-oriented programming language, like Java or C++, or an executable metamodeling language, like Kermeta [25] or XMF (eXecutable Metamodelling Facility) [6]. Executable metamodeling languages allow not only the description of the model structure (the abstract syntax), but also of the behavior. Second, each concept in the metamodel must be mapped to one class in the target implementation language. Third, the target implementation language must provide support for aspect programming, to allow inserting the code responsible for the composition semantics into the original metamodel implementation (the set of corresponding classes, responsible for model interpretation) without changing the interpreter. In this context, one decided to use Java for implementing our interpreters, together with its aspect-oriented extension, AspectJ.

Models, defined using the DSL abstract syntax concepts, are technically reified as Java classes and then interpreted. This implies that the model is created before execution, while the instances of semantic domain concepts are only created during the execution. More precisely, at design time, the modeler only needs the abstract syntax concepts for creating a model – referred to as domain specific model; he or she does not need to be aware of the concepts related to the interpretation. Models are represented, at execution, as instances of the AS classes, and are interpreted using the semantic domain. At run time, the model is simply reified as instances of the interpreter classes and then interpreted. However, during execution, the interpreter modifies/creates/deletes instances of the abstract syntax concepts, and also creates instances of the DSL concepts corresponding to the semantic domain.

4.2. Code Generation

The Eclipse mappings currently used in Mélusine environment for the vertical composition are presented in Table 2. Actually, users never see, and even ignore, that AspectJ code is generated; for instance, they do not create an AspectJ project, but simply define and generate a feature associated with a concept. A similar idea is presented in [30], where Xtend and Xpand languages are used for specifying mappings from problem to solution spaces and the code generation is considered to be less error-prone than the manual coding.

To implement horizontal relationships in AspectJ, each horizontal relationship is also transformed into an AspectJ code. The mappings towards Eclipse artifacts used for Mélusine horizontal composition are indicated in Table 3.

4.3. Codèle Tool

This section introduces Codèle, as an implementation for the composition methodology previously presented. For supporting domain composition, we have developed the Codèle toolbox, in which dedicated editors allow one to: (i) Define horizontal relationships, (ii) Use horizontal relationships to define static model composition, (iii) Use horizontal relationships to define dynamic model composition.

From this information, Codèle automatically generates AspectJ captures and the code that implements the composition strategy. Implementing horizontal relationships in AspectJ is simple. Each connection is transformed into an AspectJ code that calls a method in a class gen-

Table 2. Mapping on Eclipse artifacts for the vertical composition metamodel

Metamodel element	**Eclipse artifact**	**Elements generated inside the artifacts**
Domain	Project	Interfaces for the domain management
Concept	Class	Skelton for the methods
Behavior	Method	Empty body by default
Feature	AspectJ Project	The AspectJ aspect and a class for the behavior
Abstract service	Project	Java interface defining the service interface
Service	Project	An interface and an implementation skeleton
Interception	AspectJ Capture	The corresponding AspectJ code

Table 3. Mapping between horizontal composition concepts and Eclipse artifacts

Metamodel element	**Eclipse artifact**	**Elements generated inside the artifacts**
Domain	Project	Predefined interfaces and classes
Concept	Class	None
Behavior	Method	None
HorizontalRelationship	AJ Class and Java classes	– an AspectJ file containing the code for all the interceptions – a Java file for each instantiation connections – a Java file for each synchronization connections
Interception	AspectJ Capture	Lines in the AspectJ file for the interception, and a Java file for the connection code

erated by Codèle; users never "see" it. In practice, the code for horizontal relationships semantics represents about 15% of the total code.

Under a unified graphical interface, Codèle implements different subsystems:

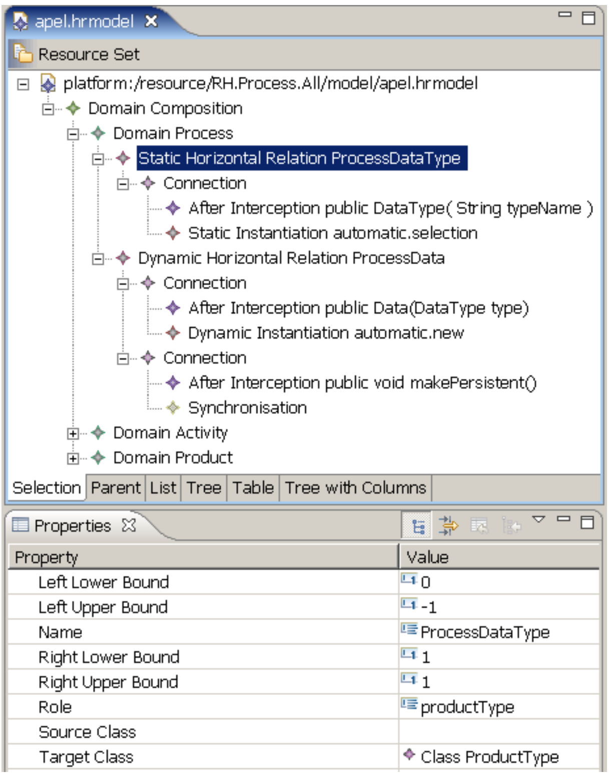
– *Relationships Editor*, which is responsible to create horizontal relationships, according to the properties presented above; see an example in Fig. 7, for defining a relationship between *DataType* from *Activity* domain, and *ProductType* from *Product* domain;

– *Captures Generator*, which generates AspectJ code, and creates a Java class in which the user can define the connection semantics;

– *Dynamic Model Composition Editor*, for dynamic link creation and life cycle;

– *Static Model Composition Editor* for the composition of two models, in their abstract form; see an example in Fig. 8.

In our example, the *DataType* – *ProductType* horizontal relationship has been selected, for which one displays the corresponding instances, like Specification in the Activity domain, and JMLSpecification or JavaFile in the Product domain. As this horizontal relationship has been declared Static, the developer is asked to provide the pairs of model entities that must be linked together, according to that horizontal



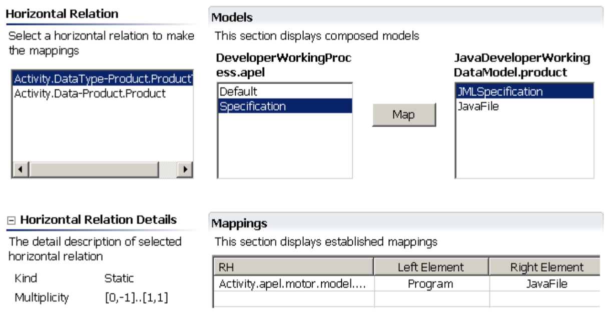
```
platform:/resource/RH.Process.All/model/apel.hrmodel
  Domain Composition
    Domain Process
      Static Horizontal Relation ProcessDataType
        Connection
          After Interception public DataType( String typeName )
          Static Instantiation automatic.selection
      Dynamic Horizontal Relation ProcessData
        Connection
          After Interception public Data(DataType type)
          Dynamic Instantiation automatic.new
        Connection
          After Interception public void makePersistent()
          Synchronisation
    Domain Activity
    Domain Product
```

Property	Value
Left Lower Bound	0
Left Upper Bound	-1
Name	ProcessDataType
Right Lower Bound	1
Right Upper Bound	1
Role	productType
Source Class	
Target Class	Class ProductType

Figure 7. Defining horizontal relationships at metamodel level

relationship. Otherwise, they would have been selected automatically, at run time. The bottom panel lists the pairs that have been defined. For example, the data type called Program in the

Activity domain is now related to JavaFile in the Product domain. The system finds this information by introspecting the models and is in charge of creating these relationships at model level.


Horizontal Relation
Select a horizontal relation to make the mappings
Activity.DataType-Product.Product
Activity.Data-Product.Product

Models
This section displays composed models
DeveloperWorkingProcess.apel
Default
Specification
Map
JavaDeveloperWorkingDataModel.product
JMLSpecification
JavaFile

Horizontal Relation Details
The detail description of selected horizontal relation

Kind	Static
Multiplicity	[0,-1]..[1,1]

Mappings
This section displays established mappings

RH	Left Element	Right Element
Activity.apel.motor.model....	Program	JavaFile

Figure 8. Defining static links at model level

Mélusine system, including the domain composition technology, was developed in 2000 and was used in a number of applications, both academic and industrial. A little less than one million lines were developed for this system, and dozens of domain compositions were performed. The work reported in this paper started with an analysis of these domain compositions, with the goal to find recurring concepts and patterns, and ended in the development of the Codèle tool. Since then, Codèle is integrated in different environments. In some environments, like FOCA [27] where domains are manually composed, Codèle is used only for model composition. In other systems, like Mélusine, Codèle is fully used, on a daily basis.

5. Evaluation of the Composition Approach

5.1. Related Works

The works on model/metamodel composition can be classified according to several criteria: the composition mechanism and the theme of research. According to the composition mechanism, these works could be split in two major categories: the heavyweight composition mechanism, which consists in model and metamodel merging [29], [23], [20]; and the lightweight mechanism, which involves establishing synchronization relationships [11] or weaving two models/metamodels, without changing their structures. The second mechanism is interesting because it is possible to compose models and metamodels and still use their existent tools.

According to the theme of research, model/metamodel composition is approached in three major areas: *Model Management, Aspect Oriented Modeling* and *Metamodeling*.

Model Management is a topic born in the MDE (Model Driven Engineering) context. This community is interested in platforms manipulating and managing models, focusing on generic operators to be applied on models, which can be divided in three groups:

- *match* [3, 4], *relate* [21], *compare* [20] – for discovering correspondences between models;
- *merge* [21], [20], [7], *compose* [3], *weaving* [7] – for integrating models and
- *sewing* [7] – for relating models without changing their structure.

Several platforms have been developed, like AMMA [28], Rondo [10], EOL [23] and MOMENT [19]. One can qualify these *Model Management* approaches as heavyweight.

Aspect Oriented Modeling (AOM) applies the separation of concern principle of AOP in the modeling phase. Weaving consists in composing aspect models to a base model. The relationship between aspect model and base model is relative. A model can be both an aspect and the base; thus, two kinds of weaving have been identified: aspect/base weaving (called asymmetric), and base/base weaving (called symmetric). The first one is borrowed from AOP and usually uses a lightweight composition mechanism, while the second one is inspired from SOP (Subject Oriented Programming) and uses a heavyweight mechanism. Theme/UML [7] is an approach merging both kinds of weaving; the composition between the base models (called subject) is done with two kinds of composition relationships: *merge* or *override*. *Merge* integrates a subject with another one, while *override* replaces an existing subject with a new one. In all cases, these strategies change the composed model structure. The aspects in Theme/UML are designed in terms of aspect templates.

Metamodelling also treats model compositions, supported by *metamodeling* environments, like XMF (eXecutable Metamodelling Facility) [6] or GME (Generic Modeling Environment) [9]. XMF has a purpose that is similar to ours – lightweight model composition consisting of composing and executing models conforming to different metamodels. This is possible through synchronized mappings, written in XSync – a specific language of XMF, based on actions. Unfortunately, the metamodels also have to be written in a specific language – XCore, which is an extension of MOF. Therefore, we would not be able to reuse our metamodels (implemented in Java) nor our models, nor use AOP technique – which is a central requirement for model and metamodel reuse.

GME environment also supports the composition of models conforming to the same metamodel (using so-called references) and to different metamodels (using union and inheritance). However, it allows the creation of a composite metamodel, which may be used for defining new models; there is no possibility to reuse the existing models "as-is" and to keep the metamodels unchanged – an important requirement for our domain composition approach.

The canonical scheme for model composition proposed in [5] uses a weaving model, consisting in correspondences between model elements. Then, several transformations based on ATL (ATLAS Transformation Language) are used for obtaining the composite model. The composition semantics resides in these transformations. The weaving model may also be extended for creating a specific composition, using AMW (Atlas Model Weaver). This facility could be used for defining our horizontal relationships; however, our purpose was to obtain a composition tool based on wizards, which is easier to learn and only contains the concepts specific for our composition approach.

5.2. Specificities for Mélusine Composition

In order to make the domain composition task as simple as possible, the metamodels presented above took into account the specificities of Mélusine domains. Consequently, the composition we realized is specific for this situation, as opposed to other approaches, which try to provide mechanisms for composing heterogeneous models in general contexts, generally without specifying how to implement them precisely.

The technique used at each composition level is different. At code level one uses AOP technique; at model and metamodel levels one establishes relationships. The main reason for this choice was to compose domains without changing their models or the associated tools and environments.

The elaboration of metamodels that support code generation in Codèle tool was possible after years of performing Mélusine domain compositions. This experience also led to the definition of a methodology for developing horizontal relationships, described in [11]. Moreover, through trials and errors, one found recurring patterns of code for defining vertical and horizontal relationships and it was possible to identify some of their functional and non functional characteristics. Codèle embodies and formalizes this knowledge through simple panels, such that users "only" need to write code for the non standard functionalities. Practice showed that, in average, more than half of the code is generated, in an error prone manner, managing the low level technical code – including AOP captures, aspect generation and so on. The user's added code fully ignores the generated one and the existence of AOP; it describes the added functionality at the logical level. Experience with Codèle has shown a dramatic simplification for writing relationships, and the elimination of the most difficult bugs; there are also some cases where the generated code was sufficient, allowing application composition without any programming.

However, many other non-functional characteristics could be identified and generated in the same way, and Codèle can (should) be extended to support them. We have also discovered that some, if not most, non-functional characteristics cannot be defined as a domain (security, performance, transaction etc.), and therefore these non-functional properties cannot be added through horizontal relationships. For these properties, we have developed another technique, called model annotation, described in [27].

6. Conclusion

The division of applications in parts can be performed by reusing large functional areas, called domains, which are primary elements for dividing the problem in parts, and atoms on which our composition technique is applied.

A domain is usually implemented by reusing existing parts, found on the market or inside the company, which are components or tools of various size and nature. We call vertical composition the technique which consists in relating the abstract elements found in the domain model, with the existing components found in the company. Reuse imposes that vertical relationships are implemented, without changing the domain concepts, or the existing components. In our approach, one develops independent and autonomous domains, which become the primary units for reuse, whose interfaces are their domain models (DSLs).

Domain composition is performed by composing their DSLs, without any change in their abstract syntax or semantics. This is called horizontal composition, defining relationships between modeling elements pertaining to the composed domains. In this way, the tools/environments in charge of editing, analyzing and executing the models, as well as the knowledge of practitioners, are kept unchanged. Tools, environments and models can be reused "as-is" and thus they can continue to be used by the existing applications that rely on them, which is a critical property in real operational contexts.

An important goal of our approach was to raise the level of abstraction and the granularity level at which large applications are designed, decomposed and recomposed. Moreover, these large elements are highly reusable, because the composition only needs to "see" their abstract models, not their implementation. Finally, by relating domain concepts using wizards, most compositions can be performed by domain experts, not necessarily by highly trained technical experts, as it would be the case if directly using AOP techniques.

References

[1] E. Barra Zavaleta, G. Génova Fuster, and J. Llorens Morillo. An approach to aspect modelling with uml 2.0. In *Proceedings of the UML 2004 Workshop on Aspect-Oriented modeling*, Lisbon, Portugal, 2004.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 2003.

[3] P. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, Asilomar, CA, USA, 2003.

[4] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, Shanghai, China, 2006. ACM.

[5] J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige. A canonical scheme for model composition. In *Proceedings of the European Conference in Model Driven Architecture (EC-MDA)*, Bilbao, Spain, 2006.

[6] T. Clark and al. Applied metamodelling – a foundation for language driven development version 0.1. Xactium, Editor, 2004.

[7] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.

[8] T. Dave. Reflective software engineering – from MOPS to AOSD. *Journal of Object Technology*, 1(4), 2002.

[9] J. Davis. GME: the generic modeling environment. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '03)*, Anaheim, CA, USA, 2003.

[10] M. Didonet Del Fabro and F. Jouault. Model transformation and weaving in the amma platform. In *Proceedings of the Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE)*, Braga, Portugal, 2005.

[11] J. Estublier, A. D. Ionita, and G. Vega. Relationships for domain reuse and composition.

Journal of Research and Practice in Information Technology, 38(4):135–162, 2006.

[12] J. Estublier, G. Vega, and A. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proceedings of the MoDELS/UML Conference*, pages 69–83, Jamaica, 2005. Lecture Notes in Computer Science.

[13] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development.* Addison-Wesley, ISBN10: 0321219767, 2004.

[14] G. Génova, C. Ruiz del Castillo, and J. Lloréns. Mapping UML associations into Java code. *Journal of Object Technology*, 2(5):135–162, 2003.

[15] J. Gray, T. Bapty, S. Neema, D. Schmidt, A. Gokhale, and N. B. An approach for supporting aspect-oriented domain modeling. In *Proceedings of GPCE.* LNCS 2830, Springer Verlag, 2003.

[16] W. Ho, J.-M. Jezequel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect-oriented UML designs. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, pages 99–105, Enschede, The Netherlands, 2002.

[17] A. D. Ionita, J. Estublier, and G. Vega. Variations in model-based composition of domains. In *Proceedings of the Software and Service Variability Management Workshop*, Helsinki, Finland, April 2007.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[19] D. Kolovos, R. Paige, and F. Polack. Eclipse development tools for epsilon. In *Proceedings of the Eclipse Summit Europe, Eclipse Modeling Symposium*, Esslingen, Germany, 2006.

[20] D. Kolovos, R. Paige, and F. Polack. Merging models with the epsilon merging language (eml). In *Proceedings of MoDELS'06*, pages 215–229. LNCS 4199, 2006.

[21] I. Kurtev and M. Didonet Del Fabro. A DSL for definition of model composition operators. In *Proceedings of the Models and Aspects Workshop at ECOOP*, Nantes, France, 2006.

[22] T. Le-Anh, J. Estublier, and J. Villalobos. Multi-level composition for software federations. In *Proceedings of the SC'2003 Conference*, Warsaw, Poland, April (2003). IEEE Computer Society Press.

[23] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *Proceedings of the International Conference on Special Interest Group on Management of Data (SIGMOD)*, San Diego, California, June, 2003.

[24] M. Monga. Aspect-oriented programming as model driven evolution. In *Proceedings of the linking aspect technology and evolution (LATE) workshop*, Chicago, 2005.

[25] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the MoDELS/UML Conference*, Jamaica, 2005. Lecture Notes in Computer Science.

[26] A. Occello, O. Casile, A. Dery-Pinna, and M. Riveill. Making domain-specific models collaborate. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Montréal, Canada, 2007.

[27] G. Pedraza and J. Estublier. An extensible service orchestration framework through concern composition. intl workshop on non-functionnal properties in domain specific languages. In *Proceedings of the NFPDML conference*, Toulouse France, 2008.

[28] T. Reiter and al. Model integration through mega operations. In *Proceedings of the Workshop on Model-driven Web Engineering (MDWE)*, Sydney, 2005.

[29] M. Sabetzadeh and S. Easterbrook. Easterbrook: An algebraic framework for merging incomplete and inconsistent views. In *Proceedings of the 13th IEEE International Requirements Engineering Conference*, pages 306–318, 2005.

[30] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Prouct Line Conference (SPLC)*, Kyoto, Japan, 2007.