

A Domain Composition Approach

Jacky Estublier
LSR-IMAG
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
jacky.estublier@imag.fr
+33 476 63 55 64

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

ABSTRACT

Domain engineering is a very efficient approach, for narrow and stable domains. Unfortunately, building a wide-scope application often involves crosscutting several domains. The paper addresses the issues related to the creation of large applications, by composing existing domains. The approach specificity stands in reusing domains, without changing or adapting them, even when they have been independently designed and implemented. The solution consists of establishing relationships between formal, executable, domain models.

Our approach relies on the research and experience gained when using UML relationships (with their implications in structural and behavioral views) and the various approaches for transforming UML models into code. Domain composition requires new kinds of relationships between concepts from different domains and a new methodology for defining and implementing them. The paper shows the issues addressed in our federation architecture and the solutions found in environment Mélusine, that implements it. On the basis of the experience achieved in developing composite domains for industrial applications, a preliminary methodology for establishing inter-domain relationships was identified.

Keywords

Model Composition, Reuse, Domain Engineering, MDE

1. Introduction

Domain engineering covers software applications where one of the primary concerns is a “systematic software reuse” [1]. A domain can be seen as a set of systems built from common assets. Most domain engineering approaches involve an initial analysis phase, where the commonalities and differences between family members are identified. This analysis is used to build a common architecture, that embodies a large part of the domain knowledge; it is implemented once and is shared by all applications in the domain, such that a new application only needs to implement its specificities. The approach is interesting if the common part represents a large fraction of the complete application, i.e. if the domain is narrow.

The main drawback of domain engineering is that it only covers a limited scope. To preserve the domain

engineering approach, an obvious solution is to compose domains. Unfortunately, the many attempts to do so have not been quite successful.

There are different approaches for domain engineering; in generative programming, a generator takes as input a feature model and generates the executable code [2]. In Domain Specific Language (DSL) approach [3] a language is defined, in which a given application is “programmed” and then compiled into executable code. Both approaches rely on language and compiler technology. Composing domains would therefore require to define extensible languages [4], that is known to be very difficult, or to compose different languages, that is known to be even more difficult, at least when using the classic language and compiler technology. Finally, building compilers and specialized programming environments is very expensive. Therefore, it is not a surprise to see that the current domain engineering approaches have failed so far to solve the issues of extending and/or composing domains.

Our work addresses domain composition from a model/meta model point of view, instead of a language/compiler point of view. We show that, if domains are formally defined by a domain model, domain composition can be simply performed through the definition of relationships between the domain models. Based on our experience gained in using the system for several years, 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 [5], is based on two main ideas:

- rely on a meta level, where the characteristics common to all the applications of a domain are gathered;
- establish relationships between elements from different domains, at all levels (meta-model, model and instances).

The paper presents the methodology adopted in Mélusine environment, that supports the development of software federations, for identifying, specifying and implementing these relationships, such that they grasp the entire semantic added for realizing the composition - that remains in this way external to the composed domains.

The elements that are common for any domain composition, those common for any application inside a domain and those specific to a particular application are

clearly separated. At the same time, the methodology points out the way structure and behaviour influence each other while passing from analysis to design, in order to treat the problems related to rendering the models executable, synchronizing the domain states, managing the relationships life cycle and assuring their persistency.

The paper is structured as follows. Section 2 presents what does it mean domain modeling; section 3 presents the principle of domain composition. Section 4 presents the method we are using when it comes to compose domains. Section 5 relates our work to other works on relationships and section 6 concludes the paper.

2. Domain models and meta-models

A domain can be defined from different points of view:

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

The ontological point of view [6] leads to the definition of classes of entities that, together, constitute the domain. The second point of view leads to a type of Domain Specific Language (DSL) in which only variations (features) can be expressed [7]. The last one leads to the definition of a domain model, that is a mixture of the other ones, and borrows its concepts and terminology from Model Driven Engineering (MDE)[8]. Our approach falls into the last category.

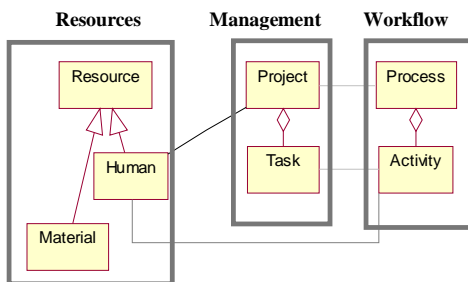


Figure 1 Domain meta-models

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 meta-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. Figure 1 shows an example of 3 meta-models, for Resources, Management and Workflow domains. This approach clearly separates the

common part, shared by all applications in the domain, which is represented through primitive language constructs, from the specific aspects of each individual application, which are defined in a model conform-to the meta-model. Our framework allows models to be mapped to various tools, that might have been developed before the creation of the domain. As often in UML, we transform the concepts identified in the meta-model into (Java) classes, operations into methods and the common behaviour into Java code. The model is also transformed into instances of the classes from the (Java) meta model. Thus, both operational perspective (the tools) and descriptive one (the model) are independently supported.

3. Relationships for domain composition

The basic rule in creating a domain is to define a meta-model, that only contains the required, high level, common concepts and to let the variability to be defined into models and tools.

The advantage is that, at meta-model level, a domain is easy to understand and manipulate; the drawback is that, being very simple, the meta-model 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 were independently defined and developed.

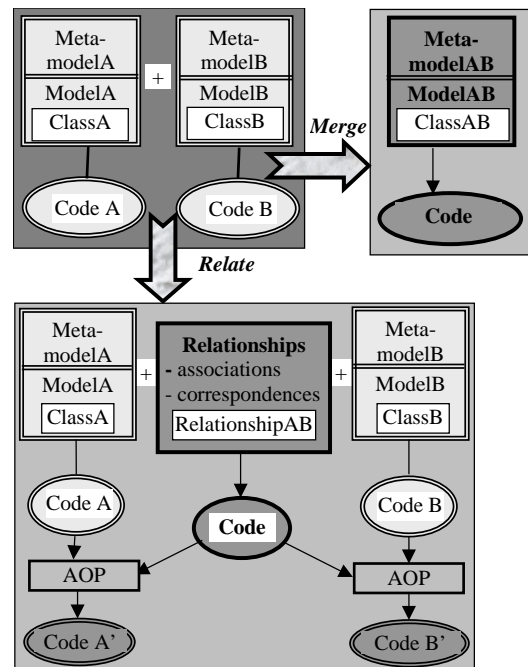


Figure 2 Model composition alternatives

Generally, the composition is treated as a merge followed by code regeneration (see Figure 2); models A and B have usually the same meta-models (e.g. UML), such that

the problem stands in merging the models. UML meta-model extensions are proposed by Clarke [9], while a Hyper/UML approach is given by Philippow et al in [10]. The problem of meta-model composition is seldom addressed in the literature, an exception being the GME environment, which allows DSL composition [11]. In contrast, our system heavily relies on model and meta-model composition.

In model merging, the original code for A and B is not reused and a new code has to be produced. The approach is reasonable only if the code can be fully generated, as in generative programming. In all the other cases, the approach does not facilitate software reuse, which is essential in domain engineering. In order to reuse code and models as they are, instead of merging, which can lead to a fairly different new model, we limit ourselves to establish relationships between the existing meta-models and between the pertaining models. Since we decided not to change the original models, these relationships must contain all the semantics added when composing the domains.

For example, in figure 1, for Management and Workflow domains, one may identify a relationship between Project and Process concepts, meaning that what is seen as a project, in the Management domain, can be seen as a process in the Workflow domain. Relationships can also be established between Task and Activity concepts and between Project and Human from the Resources domain (because a project has a manager and employees working for it). The reuse constraint requires not to change the meta-model, nor the code obtained after its (partially manually) transformation in Java. In particular, it means that the relationships and their semantics necessary for domain composition can not be transformed into new code added to the original Java source code. Nevertheless, collaboration requires navigation between domains. For that reason, the relationship semantic is transformed into “aspects” in the AOP (Aspect Oriented Programming) sense; to do so, we use our Extended Object Machine (EOM) [12] for capturing method calls and modifying the byte code.

4. Domain Composition Methodology

Composing domain, as proposed above, is an unusual task. It required us several years and many realizations before identifying rules of thumb, that help in the composition process. These rules and lessons are given here, but they are still far from a real methodology.

A federation domain has three layers:

- the domain **meta-model**, characterizing what is common for all the applications inside that domain;
- the application **model**, that conforms to the meta-model;
- the **instances** generated when executing the application.

This architecture follows the reuse principle: what is common in the domain is gathered in the meta-model and shared by all applications, such that an application is defined

(by a model) only by indicating its specificities in the domain. This is why the approach is only valuable if the domain is narrow enough, to share a substantial body of knowledge.

Given this three level architecture, domain composition through definition of relationships has to be performed at three levels also (see figure 3). However, the issues and concepts are pretty much different at each level, as explained in the following sections.

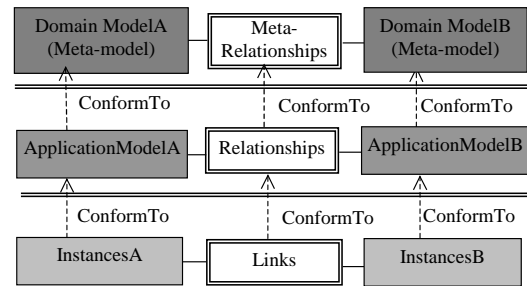


Figure 3 Relationships on federation levels

4.1 Meta model Relationships: Meta model composition

Two types of relationships can be established between concepts pertaining to two domain models (meta-models):

- *associations*, that are similar to UML associations;
- *correspondences*, that relate same or overlapping concepts.

As domain models are designed independently, they often contain similar concepts, defined in different ways, as each domain outlines the characteristics important for it. For example, the concepts Process and Project have similarities but, as they are defined 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 project is always associated with a process and vice versa. This relationship is called correspondence. The tuple {Project, correspondence, Process} implements de facto the abstract concept. More generally, a correspondence between two concepts means the identification of a new concept - implemented as a tuple {Concept1, correspondence, Concept2}.

Managing a concept in this way raises a number of problems. Indeed, each original concept rests a part of an independent domain, that uses it in a certain way. A first difficulty is to identify which are the consequences of a concept operation on its correspondent concept; more generally, what is the correspondence between a state change of in a domain and the state of the other domain. For example, operation “startProject” in the Project concept may be associated with operation “createProcess” in the Process concept, that, among other things, may set the attributes “startDate” and “launchDate”, of the created project and process, to the same value.

The second difficulty is that the abstract concept may have additional attributes and behaviour, that are not contained by the original concepts. The solution is to model the correspondence as an association class, that supports all the additional information, but with the particular semantics related to operation correspondence, for synchronizing the states of the composed domains. UML extensions for modelling these relationships were given in [13].

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

Rule 1: If two concepts are related by a correspondence and they aggregate other concepts, it is expected to exist correspondences between some of their parts too (e.g. Task – Activity correspondence, that is a consequence of the Project – Process correspondence in Figure 1).

Rule 2: It is useful to classify operations, such that to easily identify the correspondences between elements of the same categories; for example, the operations may be: constructors, set operations, get operations, relationship related methods, viewers, persistency managers ; a concept operation is likely to be related to an operation from the same category, from the correspondent concept.

The additional behaviour of the abstract concept is expressed as collaborations between domains. The issue is to analyse if the navigation between them is possible. If there are no associations or correspondences already defined, between the appropriate concepts, navigation requires the creation of a third type of relationships - *collaborative associations* - that are necessary for giving a structural support for the imposed behaviour. For example, a collaborative association is found between Activity and Human (when an activity is finished, an e-mail should be automatically sent to the manager of the project, situated in a different domain).

These meta-relationships are domain invariants and are shared by all applications in the domain.

4.2 Model Level Relationships : Model composition.

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 should define then the relationships between these models.

The conformance between a meta-model 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 meta-class and a class. It means that the (meta) relationships defined at the meta level, between two meta-classes, must be instantiated between the classes (instances of these meta-classes) 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 {Concept1, CR, Concept2}, the set of instances of Concept1 - C1 (the Concept1 extension), and of Concept2 – C2 (the Concept2 extension), one should instantiate relationships (c1-cr-c2), with $c1 \in C1$, $c2 \in C2$, $cr \in CR$.

There is no general solution for this; it depends on the CR semantics. Nevertheless, the federation supports two types of solutions :

- *automatic instantiation.* The model composition designer provides a mapping function $f_{cr}(c1) = \{c2\}$ that, provided c1, an element of C1, returns a set of elements of C2; its inverse mapping $f_{cr}^{-1}(c2)$ might also be necessary. Relationship CR is instantiated between c1 and each one of the elements returned by the function. An usual mapping technique is to define key attributes in Concept1 and Concept2 and to associate them according to the key values. By default, the “Name” attribute is supposed to be a key.
- *manual instantiation.* For each C1 element, the model composition designer has to select a correspondent c2 element explicitly.

At model level, the relationship “cr” relates two classes that have a specific behaviour and therefore, “cr” is likely to involve a specific behaviour too, to coordinate the execution of the specific operations defined in these classes. In this case, for each relationship (c1, cr, c2), “cr” holds a specific semantics. Each “cr” is a relationship class of its own.

The application designer is in charge to :

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

Indeed, “programming” an application, in our case, is limited to performing the three activities above. Only the third one involves programming. Note that, if specific semantics are required for a relationship, automatic instantiation is either impossible, or requires a formal generator.

It is important to emphasize that model composition is to be performed only once for each pair of models, 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 C1 and C2 are two data bases (e.g. {humans - CR – cars}).

4.3 Links at execution level

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

Each class may have many different instances, therefore a mapping has also to be provided. This looks like instantiating relationships at model level, but in contrast, the

instances are statically known at model level, while here, 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.

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

- if a mapping function is provided, the instantiation is automatic;
- if a mapping function can not be defined, the *user* is invited to manually select which element(s) of C2 are to be linked.

Note that the mapping function either returns an existing element, or creates an element. For example, creating a project involves the creation of the corresponding process, while creating an activity involve selecting an existing user. In the latter case, there is no automatic mapping, the team leader will be asked to select who will do the job.

The above discussion has fundamental consequences :

- Instances do not have specific behaviour, nor the links between them; no programming is required for relationships instances.
- Models must be executable.

The first point means that the user never has to define supplementary behaviour, but only specify the corresponding instances (if no mapping is provided) or even do nothing if they are automatically determined. The latter point is very important because, to create links between instances at execution time, one should have the (Java) instances for the entities defined in the models. This defeats the classic approaches, where models are compiled and executable code is generated (see Figure 2). Conceptual composition requires model execution.

4.4 Implementation issues

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

4.4.1 Meta classes, classes and instances

We have made so far the hypothesis that meta-models contain meta-classes and models contain classes, that can express new abstractions and specific behavior. Fortunately, this is often not the case, because the domain engineering approach seeks to push at meta-model level most of the domain semantics, if not all. Experience shows that most domain specific meta-models do not provide any way to define new abstractions and therefore, models do not require any programming at all¹.

¹ In a study it has been found that only 15% provided user defined types and only one third of them provide user defined functions [26].

This is very important for two reasons. From a human perspective, the application designer defines a new application without 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 do not provide meta-classes, a meta-class/class implementation is not easy and requires some workaround. In our V3 implementation, the classic solution was adopted and model classes were generated. Currently, instead of meta-classes, the meta-model contains “class type” concepts; in the model, the new abstractions are not specific classes, but instances of these “class types”; at execution, instances are ontologically related to the corresponding type instances. Doing so, models only contain (Java) instances and the relationship between a model and its instance is based on copies and clones.

Distinguishing types also provides clues for the domain designer, on how to define correspondences.

Rule 3: If a domain concept represents a type, it should only be related to a type from the other domain.

Rule 4: If two types are related, one should search for correspondences between the concepts having those types.

4.4.2 Implementing relationships

Relationships are classically represented by class attributes, but since we are not “allowed” to change the original classes, these attributes are not added in the source code, but instead, inserted in the class byte code, using our AOP machine. The relationship semantic is represented as a specific association class. So, the differences between associations, correspondences and interaction support is only useful for analysis and for the domain composition designer, because then, they are all implemented with aspects.

Each relationship is designed as an association class, that is designed according to the type of the meta-relationship to which it conforms (association, correspondence, collaborative association) giving support for the implicit interactions that they should allow and for managing link life cycle. The association class is then implemented by an aspect, inheriting a predefined relationship class (aspect class); this design decision is valid for all relationships, independent of the type identified at analysis. In particular, life cycle 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 “createLink” 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 between 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 Melusine relationships between domains is given below.

Table 1. Relationship methods in Mélusine

Relationship methods	Link Methods	
	LifeCycle	Persistency
createRelationship	createLink	saveLink
deleteRelationship	deleteLink	recoverLink
	navigateLink	

5. Related works on relationships

Our collaboration with industry led us to realize 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.

A first issue is the problem raised by UML standard, where each link is an instantiation of the association. Stevens [14] makes the distinction between associations expressing structural relationships and those resulting from collaboration links, expressing a communication, behavioural relationship – similar to the collaborative associations from the federation. However, Génova [15] claims that not all associations should appear in a class diagram. Based on the law of Demeter, those that are “friends” and know each other do not need to be explicitly connected. Our approach actually conforms to the new law of Demeter for concerns [16], such that objects from different domains (concerns) need a structural support (an association) for interacting. Similar approaches based on designing the interactions with aspects are given by Blay [17], or by Lieberher, with aspectual collaborations [18] and the mechanism for establishing transversal associations [19].

Relationship semantic is very important for transforming it into code and for reverse engineering also. In [20], Harrison separates the semantics of associations from their representation and implementation, for mapping them to Java. Guéhéneuc 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 [21]. For similar reasons, Gogolla gave a decomposition of qualified associations, aggregations and compositions into simple associations, enriched with OCL constraints [22], transforming complex UML concepts into basic ones, that are easier transformable in code. The semantics is captured in the federation in the meta-relationships, with the particular semantic for the correspondences – necessary for composing the domains.

The support found in Melusine 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 [23], that 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 [24], where associations are supported by assessor, mutator and auxiliary methods related to relationship state and definition, in a 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 [25] one can notice that other languages, like ASL (Action Specification Language) and AL (BridgePoint Action Language) 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 Melusine environment.

Table 2. Action languages

JUMLA			
Assessors Methods	Mutator Methods	Methods for state	Methods for definition
test linkexistence 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		

6. Conclusions

The domain composition presented here is based on a rigorous domain modelling, that leads to a structure in three layers: meta-model, model, instances. This architecture is necessary for reusing already implemented domains, without performing any change inside them. Thus, the composition is defined through establishing relationships between concepts belonging to the composed domains and using an AOP machine to connect the related classes. This approach involved:

- the definition of a new kind of relationships, not found in UML;
- a new methodology for analysing and designing the relationships for domain composition;
- a new technology, for relationship instantiation at the three levels.

The practice has shown that many simplifications are possible, due to the fact that the semantic of the composition is captured in the relationships from the meta-model level.

This has a strong influence on methods and tools used for federations design and implementation.

After several years of evolution, Mélusine 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.

References

- [1] M. Simos, Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions, *STJA'97 Conf. Proc.*, Technische Universität Ilmenau, Thüringen 1997, pp. 126-132
- [2] Czarnecki, K. and Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Boston, MA, U.W. (2000)..
- [3] D. S. Wile., Supporting the DSL Spectrum, *Journal of Computing and Information Technology*, CIT 9, 2001 (4) 263-287.
- [4] Bravenboer M., Visser E. Concrete Syntax for Objects. Domain-Specific Language Embedding and Assimilation without Restrictions. *19th ACM SIGPLAN conference*. Vancouver, Canada. October 2004.
- [5] T. Le-Anh, J. Villalobos, J. Estublier. 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*, April 2003.
- [6] J. Luoma, St. Kelly, J.-P. Tolvanen, 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 2004,
- [7] R.A. Falbo, G. Guizzardi, K.C. Duarte, An ontological approach to domain engineering, *Proc. of the 14th Int. Conf. on Software Eng. and Knowledge Eng.*, Ischia, Italy, 2002, ISBN:1-58113-556-4, pp. 351 – 358
- [8] Stuart Kent, Model Driven Engineering, *IFM 2002*, volume 2335 of LNCS. Springer-Verlag, 2002
- [9] Clarke S., Extending standard UML with model composition semantics, *Science of Computer Programming*, pp. 71-100, Elsevier Science, July 2002
- [10] Philippow I., M. Riebisch, K. Boellert, The Hyper/UML Approach for Feature Based Software Design, *the 4th AOSD Modeling With UML Workshop*, Oct. 20th 2003, San Francisco, CA
- [11] A. Ledeczki, M. Maroti, G. Karsai, G. Nordstrom, Metaprogramable Toolkit for Model-Integrated Computing, *Engineering of Computer Based Systems (ECBS)*, pp. 311-317, Nashville, TN, March, 1999
- [12] F. Duclos, J. Estublier, R. Sanlaville “Separation of Concerns and The Extended Object Machine.” Submitted to Journal Advise. [http://www-](http://www-adele.imag.fr/Les.Publications/BD/ADVICE2004Est.html)

Acknowledgements: We are grateful to the whole Adele team that designed and developed the Mélusine federation environment, most notably J. Villalobos, T. Le. This work is partly supported by a Marie Curie Intra-European Fellowship, within the 6th European Community Framework Programme.

[adele.imag.fr/Les.Publications/BD/ADVICE2004Est.html](http://www-adele.imag.fr/Les.Publications/BD/ADVICE2004Est.html)

- [13] J. Estublier, A.D. Ionita, Extending UML for Model Composition, *Australian Software Engineering Conference*, 29 March – 1 April, Brisbane, Australia
- [14] Stevens P., On the interpretation of binary associations in the Unified Modeling Language, *Journal on Software and Systems Modelling*, **1**, 1, pp. 68-79 (2002)
- [15] Génova, G., J. Llorens, J. Fuentes, UML Associations : A Structural and Contextual View, *Journal of Object Technology*, **3**, 7, July-Aug. 2004, pp. 83-100
- [16] Lieberherr, K.J., “Controlling the Complexity of Software Designs”, *ICSE 2004*
- [17] Blay-Fornarino, M., A. Charfi, D. Emsellem, A-M. Pinna-Dery, M. Riveill, Software Interactions, *Journal of Object Technology*, **3**, 10, 2004, pp. 161-180
- [18] K. Lieberherr, D.H. Lorenz, J. Ovlinger, Aspectual Collaborations: Combining Modules and Aspects, *The Computer Journal*, 2003, September, Vol 46, NR 5, pp. 542-565
- [19] K. Lieberherr, M. Wand, *Navigating through Object Graphs Using Local Meta-Information*, Tech. Rep. NU-CCS-2001-05, Northeastern University, May 2001,
- [20] Harrison W., Ch. Barton, M. Raghavachari, Mapping UML Designs to Java, *OOPSLA'00*, 10/00 Minneapolis, MN, USA, PP. 178-188
- [21] Guéhéneuc Y-G., H. Albin-Amiot, Recovery Binary Class Relationships : Putting Icing on the UML Cake, *OOPSLA'04*, Oct. 24-28, 2004, Vancouver, British Columbia, Canada
- [22] Gogolla, M., M. Richters, Equivalence Rules for UML Class Diagrams, *UML'98: Beyond the Notation – International Workshop*, 1998
- [23] Mellor St.J., M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison Wesley, 2002
- [24] Génova G., C.R. del Castillo, J. Llorens, Mapping UML Associations into Java Code, *Journal of Object Technology*, **2**, 5, Sept., Oct. 2003, pp. 135-162
- [25] *OMG Unified Modeling Language Specification*, Version 1.5, Object Management Group, March 2003, <http://www.omg.org/technology/documents/formal/uml.html>
- [26] S. Tibault. Languages Dédiés : Conception, Implémentation et Application. PhD thesis. October 1998, Rennes, France.