Composition rules for PIM reuse

Salim Bouzitouna[1] and Marie-Pierre Gervais[1, 2]
[1]Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott, F-75015 Paris
{Salim.Bouzitouna, Marie-Pierre.Gervais}@lip6.fr
[2]Université Paris X

Abstract: *In order to reduce the cost of the evolution of companies' applications, this evolution should be led in a systematic way by reusing existing applications. In MDA approach, this should be done by the reuse of PIM and PSM of the concerned applications. Indeed, the reuse of models exploits those that already exist and which have been checked and maintained. It aims to construct new applications by composing, extending or modifying existing distributed applications. To this end, we propose a new initiative of distributed applications' construction by reusing models in MDA approach. Our initiative is based on two principal points: the expression of the reuse of PIM and the automatic generation of glue binding their corresponding PSM from this expression. In this paper we focus on the first point which is the expression of the reuse in terms of composition, extension and modification of PIM.*

1. Introduction:

To make the migration of company's applications towards new platforms easier, MDA approach [OMG 03] recommends a well-delimited separation between business aspects and implementation details aspects of an application. This separation is expressed via two models: PIM (*Platform Independent Model*) which specifies business aspects of a distributed application and PSM (*Platform-Specific Model*) which specifies implementation details on a specific platform. However, we can observe that the merge and reorganization of companies requires the evolution of their applications. For instance, in the case of fusion of two companies, this evolution can be expressed in terms of composition of applications. It can be also expressed in terms of extension or modification if the functionalities of existing applications are respectively extended or modified. In order to reduce the cost of these evolutions the reuse of existing applications is essential. In MDA approach, this reuse consists in reusing PIM and PSM of the existing applications.

Many approaches are interested in the problem of reuse. If we consider known levels such as MDA PIM, MDA PSM or code, none of the current approaches deal with the reuse in these levels. The majority of the approaches provide reusability in terms of code and not of abstract models. Examples of such approaches are *Subject Oriented Programming* [Harrison 93], *Aspect Oriented Programming* [Kiczales 97] or *Component Oriented Programming* [OMG 99] [Sun 99]. Only few approaches provide reusability of abstract models, similar to PIM, such as the *Subject Oriented Design* [Clarke 01]. Moreover their means carry out direct changes on the reused models. This does not guarantee a good traceability of the evolution of the reused models.

We present a new initiative based on the reuse of models for the construction of new distributed applications in MDA approach. ***Our initiative is based on two principal points: the first one is the expression of the reuse of PIM in terms of composition, extension, and modification while the second one concerns the automatic generation of the glue from this expression.*** This glue binds PSM corresponding to the reused PIM. As it depends on the platforms considered for PSM, it could be considered as the component that will be used for the assembly of these PSM. Figure 1 shows the idea of composition of two applications.

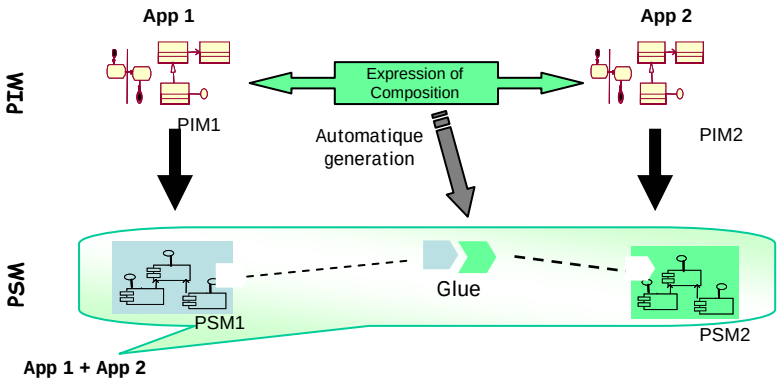

Figure 1. Our approach for the composition of MDA applications

Through this initiative, we propose a solution which deals with the reuse of MDA applications on all levels. It allows the reuse of models that already exist and which have been checked and maintained. At PIM level, the expression of the reuse of models is only used to describe how they are composed, extended or modified, but does not change them. This allows a suitable stability of reusable PIM by keeping good traceability of their evolution - which is the basic principle of MDA approach - since it supposes that the PIM of a given application remains stable. At PSM level, the glue allows the corresponding PSM to be kept unchanged, which therefore makes it possible to exploit the codes corresponding to these PSM in new applications with no change. Moreover, it allows to use the same PSM to build several new applications according to the intentions' of reuse expressions of PIM.

In this paper we focus on the first point of our initiative which is the expression of the reuse of PIM. For that, we study the different types of reuse of PIM. From these, we then define a set of rules for reuse expressions for composition as well as for extensibility of PIM. The last section concludes the article and presents some future works.

2. Integration of the reuse of PIM in MDA approach

2. 1 Expressions of PIM

PIM considered in MDA approach are expressed in a well-defined precise modeling language. This describes the structural aspect as well as the behavioral aspect of the application. The OMG recommends within the context of MDA approach the use of UML language [OMG97]. In this article, we are particularly interested in UML class diagram and UML collaboration diagram. These diagrams are very appropriate for expressing the structure and behavior of an application respectively. Using these two diagrams, we propose to describe an application independently of any platform. The class diagram represents the set of entities interacting in a given application, as well as the relations between them. It also expresses the progress of different operations defined by the entities used in the collaboration diagram. As UML recommends the gathering of these diagrams in packages according to application's functionalities they describe, we consider that applications are packaged.

2.2 Reuse of PIM

Generally, the reuse of a software unit can be expressed by several intentions, illustrated in figure 2:

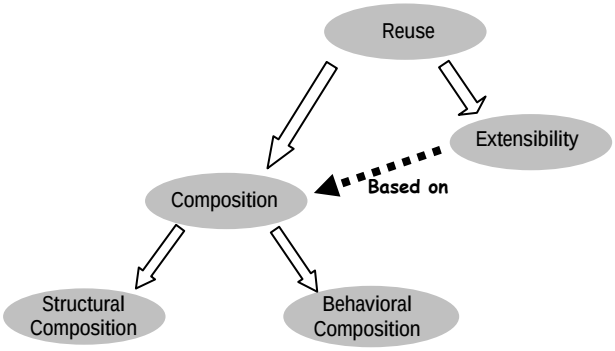

Figure 2. Different intentions of unit's reuse.

A first aspect of units' reuse is composition. It expresses the way in which this unit is assembled with others in order to form a new application. We consider two types of compositions: *structural composition* and *behavioral composition.* The structural composition aims at modifying elements of the units, while *the behavioral composition* aims at the expression of interactions between the various operations of the units. In our context the units correspond to PIM.

Applying structural composition at PIM level consists in focusing on UML class diagrams. The composition consists of merging different elements belonging to these models, such as classes and attributes. This merge consists in putting these elements together. However in order to avoid redundant elements, the elements which correspond to the same entity (*the classes' elements for example*) or the same property (*the attributes' elements for example*) will be represented by only one element among them.

We also consider modification as a form of structural composition. Basically, it consists in defining all the changes to be brought on a PIM, in another separate model. Then, it is a matter of replacing elements of the first model by those defined in the second one.

The behavioral composition is related to UML collaboration diagrams which correspond to the various PIM. It describes the interactions between the operations defined in the classes of these models. This composition consists, for example, in combining a set of operations belonging to different models by coordinating them in a given order.

The second aspect of units' reuse is extensibility. This consists in adding new functionalities to units**.** Most of reuse approaches recommend adding a new component such as *Subject Oriented Programming* [Harrison 93], *Aspect Oriented Programming* [Kiczales 97]. Their idea consists in placing all functionalities to be added in a new unit, and then composing it with the original units. Similarly to extend PIM functionalities, we propose to specify the new functionalities in a separate model and then compose them with the original model. This approach has many advantages. It will allow to keep a good traceability of the evolution of PIM. Furthermore, it allows to apply several extensions to same the PIM, which do not depend on others. We thus note that the composition of models also encompasses extensibility.

To express structural as well as behavioral composition, we define a set of rules. As these rules, applied on PIM, are abstractly defined, we call them *patterns of composition*. Thanks to these patterns, a designer can model the application he wants to build, modify or extend. However, contrary to the major trend, we do not advocate the elaboration of new PIM. Actually, many approaches, such as *Subject Oriented Design* [Clarke01], propose to apply rules on existing models in order to obtain new PIMs that replace current ones. In this way, latest changes are carried out on the current models. This does not guarantee a good traceability of the evolution of the reusable models. This compromises the basic principle of MDA approach which supposes that the PIM of a given application remains stable. To face these disadvantages, we propose to keep PIM unchanged when they are reused. Indeed, our rules do not apply to the PIM source model for building a new PIM. They are only used to express the composition between existing original models. The resulting model is composed of PIM original models and the newly defined composition rules. Figure 3 compares our step with those of other approaches.

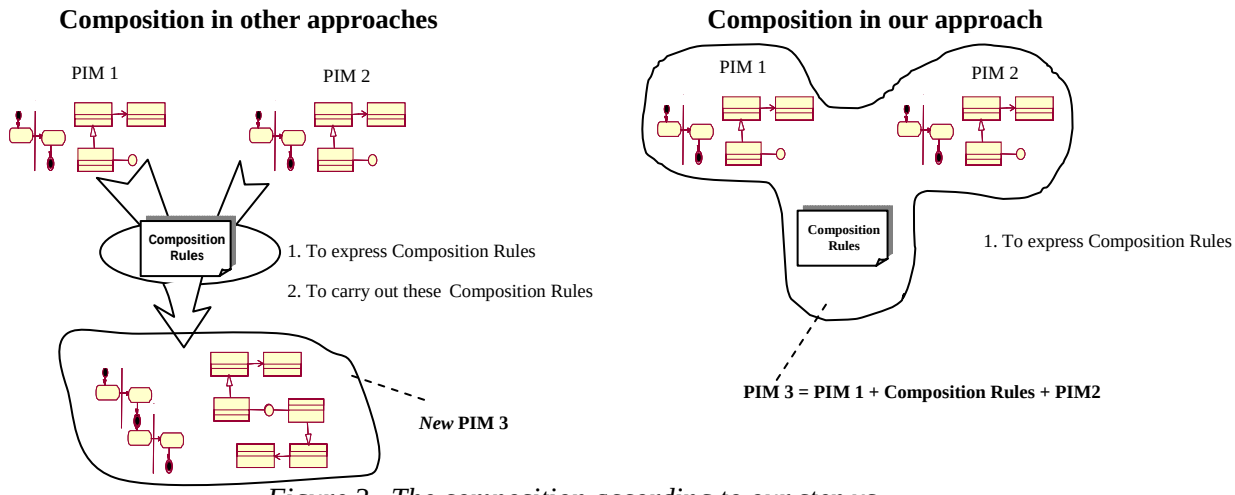

Figure 3. The composition according to our step vs the composition in the other approaches

Our rules of composition are defined as being composition patterns. This approach enables their later implementation by using any language that allows parsing models. This is proposed by many model transformation languages. To this end, we consider in the near future the use of MOF QVT [OMG 02] suggested by OMG. The choice of such a language allows compliance with OMG standards.

The set of the mentioned rules are presented in the next section.

3. Rules for PIM composition

To identify different compositions between PIM, we studied application construction approaches aiming at conceptual model's reuse as well as and those aiming at the code reuse such as [Clarke 01] [Van 99] [IBM 03a] [IBM 03b] [AspectJ 03]. We examined more particularly the means and techniques which they offer to make the composition of their component units. This enabled us to define a set of composition rules which allow to specify many types of composition of PIM.

For structural composition, these rules allow to identify more precisely, in models to be composed, different packages to be integrated, as well as elements that specify the same concept and which thus must be combined. For behavioral composition, these rules specify combination of operations defined in models to be composed. This combination consists in running all these operations when one of them is activated. However, control structures can be defined to modify the behavior of this run. We classified the rules which we defined in the three following categories.

3.1. Correspondence rules

Correspondence rules establish relation between elements (*packages, classes, operations, attributes)* of the models which will be later composed. These elements must be of the same type, and specify the same concept, but each element belongs to its own model. Correspondence rules do not specify how these elements can be combined. This is carried out by other rules which are defined in the second category.

Contrary to the *Subject Oriented Design* [Clarke 01], or *Subject Oriented Programming* [Kiczales 97], all correspondences must be expressed explicitly through correspondence rules. Elements having the same name in different models are not necessarily in correspondence. This avoids implicit compositions which are not wanted by the designer.

The following rule has been defined for expressing the correspondence between several packages:

- *CorrespondPackages [package1, package2…]*

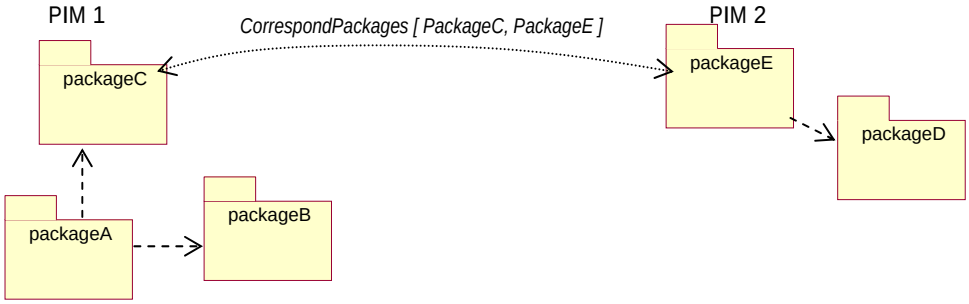

Figure4. Expression of correspondence between two packages

Figure 4 shows an expression of correspondence between two packages, each one belonging to separate model.

The expression of correspondence between packages is insufficient to express the composition between two models. We also need to specify the correspondence between their elements. This correspondence can be related to their sub-packages. In this case, it will be expressed with the same *CorrespondPackages* rule. On the other hand, it may be related to the classes of the elements. For this case, we define the following rule to express such correspondence:

- *CorrespondClasses [package1.Class1, package2.Class2…]*

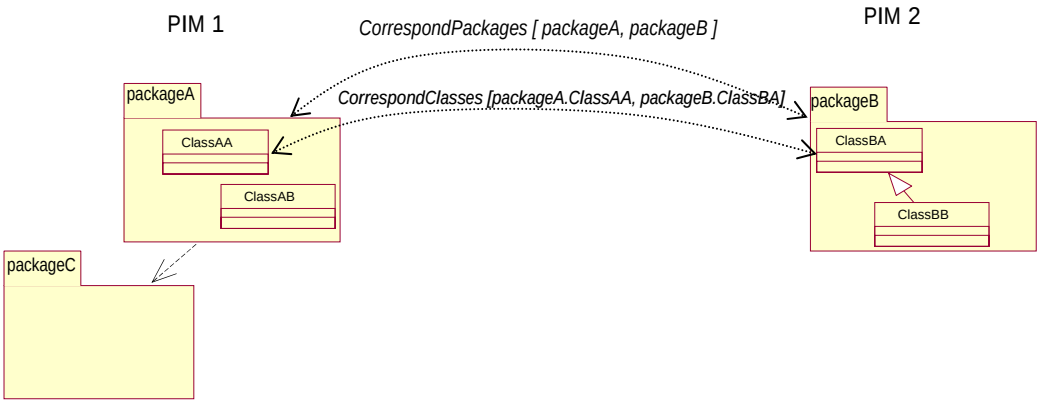

Figure 5. Expression of correspondence between two classes

Figure 5 shows the expression of correspondence between two classes: *ClassAA* defined in *packageA* and *ClassBA* defined in *PackageB.* These classes represent a priori the same entity. Note that this correspondence can be specified only if the correspondence between packages in which these classes are defined is also specified.

We can also express correspondences between attributes and operations defined in classes which have already been put in correspondence. Correspondence between attributes means that they represent the same property. Likewise, correspondence between operations of classes means that they aim at the same processing but they may perform it differently. For expressing these two types of correspondences, we propose the following rules.

Correspondence rule between the attributes:

- *CorrespondAttributes [package1.Class1.Att1, package2.Class2.Att2…]*

Correspondence Rule between the operations:

- *CorrespondOperations [package1.Class1.Op1, package2.Class2.Op2…]*

3.2. Combination rules

Combination rules are used to express the way in which composition is carried out between a set of elements (packages, classes, operations). These elements should be put beforehand in correspondence. Although a correspondence between a set of elements means that these elements represent the same concept, each one must define its proper sub-elements to specify this concept, according to its application. Thus, the composition of elements put in correspondence consists in unifying their sub-elements.

If there is a correspondence between two sub-elements, only one among them will have to be kept in their union. This is indicated by an expression of combination rules unifying elements which contain them. This indication is defined by a priority associated with each parameter of a combination rule. However, if new combination rules are defined between these sub-elements, they will cancel the priority defined between elements which contain them.

In addition, we regard the composition of a set of operations as being the execution of one or more operations in a given order. The operations to be executed as well as their order are defined using control structures which are specified in the combination rules of operations. These control structures correspond to conditional processing such as *if then, switch,* or iterative processing ones such as *for, while.*
To express a combination between many packages, the following rule is defined:

- *JoinPackages [package1, package2…]*

This rule expresses the union of classes (sub-elements) defined in each package *package1, package2…* In this union, classes which are in correspondence are represented by only one class, which is defined in the package with the greatest priority. This priority is assigned to each parameter of this rule, and corresponds to its order of appearance. Thus, classes defined in *package1* have more priority than those defined in *Package2* and so on.

We can also express combination between classes. They must be put in correspondence beforehand. To express this combination we define the following rule:

- *JoinClasses [package1.Class1, package2.Class2…]*

If a combination rule is expressed between *package1* and *package2*, a priority is assigned between their elements and thus between *Class1 and Class2.* By defining the combination rule above, the priority between these two classes are redefined. Like in a *JoinPackages* rule, the order of appearance of *JoinClasses* rule parameters defines their priorities. This defines the priority between sub-elements of classes placed in these parameters.

JoinClasses rule described above express the union of sub-elements in terms of operations and attributes defined in classes *package1.Class1 package2.Class2.* In this union attributes which are in correspondence are represented by only one attribute defined in *Class1*. Conversely, operations which are in correspondence are maintained while unifying their processing. This consists in executing all these operations when one of them is activated. The execution is carried out according to the order of priorities. Therefore, the execution of operations of *Class1* will precede the execution of that of Class2.

However, we can express the execution process of operations which are in correspondence differently from the one imposed by combination rules defined between their classes. This process may express the execution of some operations under certain conditions. It may also express the execution of one or more operations several times. To this end, we define a combination rule of operations. This rule introduces an execution process of these operations into a new operation which we call *ControlOperation.* It expresses the execution process of operations by using control structures such as *if then, switch, for* etc. Combination rule of operations is defined as follows:

- *JoinOperations[ControlOperation, package1.Class1.Op1, package2.Class2.Op2…]*

3.3. Replacement rules

Replacement rules are used to express updates of elements defined in a given model. These elements can be packages, classes, attributes or operations. An update of an element consists in replacing it by a new element of the same type, i.e. a class can be replaced only by one class, idem for operations and attributes. The definition of new elements instead of the updating of existing ones offers a good traceability of the evolution of the models.

Thus, we recommend to specify all updates of an existing model, in a separate model which we call substitute model. This one defines all new elements which will replace those defined in the original model. Therefore, it is also necessary to establish correspondences between the elements to be replaced in the original model and those of the substitute in model. This will allow the identification of the relation (*source element, substitute element*). Thus, for expressing replacements we define a set of rule which we present as follows:

- *OverridePackage [sourcePackage, updatePackage]*

This rule expresses a replacement of elements defined in *sourcePackage* by their correspondents defined in *updatePackage*. Elements defined in *updatePackage* which do not have correspondents in *sourcePackage* will be added in this one.

- *OverrideClass [package1.Class1, package2.Class2]*

This rule expresses that properties of *Class1* replace those which correspond to them in *Class2*. These properties are considered in terms of attributes and operations. Thus, if we want to replace an attribute or an operation of a given class, it is necessary to define a new class which specifies new attributes or new operations. This happens because in UML model, we cannot define an attribute or an operation apart from a class.

Generally, the rules defined in the three categories presented above can be combined. This makes possible to express the combination of two or several models while replacing some elements of the original models by elements of other models. To this end, it will be necessary to first use correspondence rules in order to define the relationship between elements that can be further combined or updated in models. Then, combinations or replacements between should be expressed by using combination or replacement rules.

4. Conclusion and future works

In this paper we have presented a solution to face the evolution of distributed applications in MDA approach. We propose in this solution the reuse of already established PIM and PSM of these applications. This solution is based on two main points: the expression of the reuse of PIM, and the generation of glue which binds their corresponding PSM. This solution is particularly useful for the reuse of existing MDA applications, in terms of composition and extensibility, without changes of their PIM and PSM.

This paper covers the first point of our solution which is the expression of the reuse of PIM. A few approaches found in the literature also propose the reuse of abstract models similar to PIM. However, the means they offer introduce direct changes on the reusable models. This compromises the basic principle of MDA approach which supposes that the PIM of a given application remains stable. Considering these observations, we have proposed a solution based on the expression of PIM reuse. To this end, we have defined three categories of composition rules: *correspondence rules*, *combination rules* and *replacement rules* which allow the expression of different intentions for reusing of PIM, considered in UML.

This article summarizes the first part of our proposal. We are currently working on its extension and improvement by considering the following parts:

- Refinement of the reuse rules we have defined. Different types of reuse in terms of composition, extension and modifications could be specified. For example, we aim at defining composite rules which combine those defined in the various categories (correspondence, combination and replacement). This will help the designer to express composition, extensibility and modifications of PIM.

- Identification of the relation between the expression of the PIM composition and its mapping on PSM i.e. the so-called glue. For this we are considering specific platforms such as CCM [OMG99] or EJB [Sun 99].

- Development of the glue generation tool. This tool consists of two parts: an analysis part which examines the set of input rules to identify the glue to be generated, and a generation part that effectively generates the identified glue. The choice of having two parts allows the generation of glues for different platforms.

References

[AspectJ 03] Eclipse:AspectJ Team, "The AspectJ™ Programming Guide", http://eclipse.org/aspectj/.

[Clarke 01] S. Clarke, "Composition of Object-Oriented Software Design Models", PhD thesis, School of Computer Applications, Dublin City University, January 2001.

[Harrison 93] W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proceedings of OOPSLA'93, ACM Press SIGPLAN, Washington, USA, pp. 411-428, October 1993.

[IBM 03a] IBM Research: Subject-oriented Programming, "Support for subject-oriented programming in C++ on IBM VisualAge for C++ v. 4", http://www-3.ibm.com/software/awdtools/vacpp/version4/.

[IBM 03b] IBM Research: Subject-oriented Programming, Group: "Hyper/J™: Multi-Dimensional Separation of Concerns for Java™ ", http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda C. Lopes, J. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In Proc. of ECOOP, pp. 220–242, 1997.

[OMG 02] OMG, Request for Proposal MOF2.0 Query /Views /Transformations, ad/2002-04-10, www.omg.org, April, 2002.

[OMG 03] OMG, Model Driven Architecture Guide version 1.0, Document Number : omg/2003-05-01. May 2003.

[OMG 97] OMG "Unified Modeling Language Specification v1.1" TC. Document ad/97-11-03. OMG. 1997. http://www.omg.org

[OMG 99] OMG "CORBA Component Model Volume 1". TC Document ad/99-01-01 OMG 1999. http://www.omg.org

[Sun 99] Sun: EJB, "Entreprise JavaBeans", http://java.sun.com

[Van 99] G. Vanwormhoudt, "CROME : un cadre de programmation par objets structurés en contextes", PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, 1999