Implementing UML Association, Aggregation, and Composition. A Particular Interpretation Based on a Multidimensional Framework*

Manoli Albert, Vicente Pelechano, Joan Fons, Marta Ruiz, and Oscar Pastor

Department of Information Systems and Computation
Valencia University of Technology
Camí de Vera s/n (46022) Valencia (Spain)
{malbert,pele,jjfons,mruiz,opastor}@dsic.upv.es

Abstract. This work presents a code generation process that systematically obtains the implementation of the UML association, aggregation and composition concepts in the context of the OO-Method (an OO automated software production method). A multidimensional framework, which identifies a set of basic properties, allows us to characterize association relationships in the OO conceptual modelling. By applying this framework, we provide a particular interpretation of the UML association, aggregation and composition concepts for the OO-Method. Once we have defined a clear semantics for these concepts, we introduce a code generation strategy that obtains the implementation of these abstractions depending on the value of the framework dimensions. This strategy can be applied to current OO development methods in order to systematize the software production process in model-driven approaches.

1 Introduction

Structural approaches to OO Conceptual Modeling propose two main steps for the construction of a conceptual schema: (1) finding classes, their attributes and operations and (2) identifying relationships between these classes. It is generally agreed in OO conceptual modeling [3] that relationships can be classified in three types: classification, generalization and association/aggregation. The association/aggregation is one of the most relevant and useful relationships. However, its semantics differs among the OO conceptual modeling approaches. The UML standard [4] does not solve this problem since the semantics that provides for association and aggregation concepts is ambiguous. This is discussed in [5] and is recognized in the RFP (OMGs Request For Proposals) to propose the new UML version 2.0 [7]. Due to this ambiguity, several approaches ([1], [2], [6], [8], [9], [12], [13], [14], [15]) have arisen in the last few years trying to define precise models for the association/aggregation relationship. But currently, a consensus for defining the semantics of these abstractions does not exist. Due to this lack

* The work reported in this paper has been funded by the CICYT project under grant TIC2001- 3530-C02-01 and the Valencia University of Technology, Spain.

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 143–158, 2003.

of consensus, if we attempt to define a process for implementing abstractions of this kind in a model-driven software production method (as our OO-Method [10] approach does), we must define a precise semantics for these abstractions.

This work discusses a particular semantic interpretation of the UML association, aggregation and composition concepts. This interpretation is obtained by applying a multidimensional framework that identifies a set of structural and functional properties (what we call dimensions) that allows us to unambiguously characterize the association/aggregation relationships. Once we have defined these conceptual abstractions, we propose a systematic method for implementing them. This method determines the OO software representation of the association/aggregation relationships according to the values of the dimensions. Our main contribution is an attempt to cover the need for improving OO methods providing systematic transformation to the design.

The paper is organized as follows: section 2 presents the multidimensional framework that we have built for identifying the properties that allow us to characterize every kind of association/aggregation relationship. Section 3 proposes a particular interpretation of the semantics of the UML association, aggregation and composition concepts. This interpretation uses the multidimensional framework presented in section 2 to define its properties. Section 4 presents a systematic approach to the implementation of the proposed association, aggregation and composition relationships. This section clearly states how the values of the dimensions presented in the framework directly influence the implementation (structure and functionality) of the relationship. Finally, we present some conclusions and further work.

2 A Multidimensional Framework

In this section, we present a multidimensional framework that identifies a set of properties which are adapted from different OO modeling methods. These properties (or dimensions) allow us to characterize association/aggregation[1] relationships in a conceptual schema.

The framework is based on an analysis process in which we have studied different approaches that analyze the association/aggregation relationships ([1], [2], [6], [8], [12], [13]). We have selected some properties from these approaches that are simple (to build an intuitive model), precise (to avoid ambiguities) and that have some influence on the software representation of the relationship (to avoid superexpressiveness). These properties should be expressive enough to ensure that the relevant characteristics of the association/aggregation relationships can be captured at the conceptual level. Some of these dimensions are used by other authors with different terms. For instance, the dimensions "delete propagation" along with the "multiplicity" defined in this work, have an interpretation which is similar to the "existence dependency" from [11]. Also, the dimension

[1] The association/aggregation relationship includes composition, which is considered a type of aggregation.

"multiplicity" that we define (similar to the way UML [4] does) is used for representing the same semantics as the property "mandatory/optional" from other approaches [6], [13].

This framework identifies the structural and behavioural properties of the the association/aggregation relationships. Thus, it provides a set of dimensions through which different kinds of association/aggregation relationships (present in current OO Models) can be categorized. This framework helps us to understand the essential characteristics and the semantics of the association/aggregation relationships, independently of the terms used for naming them.

The following subsections briefly introduce the framework dimensions. For each one, we define its intended semantics in an intuitive way, provide a nomenclature for referring to it throughout the paper, identify the element of the association that the dimension is applied to, introduce its possible values, and show the UML attributes that have close semantics to our proposed dimensions.

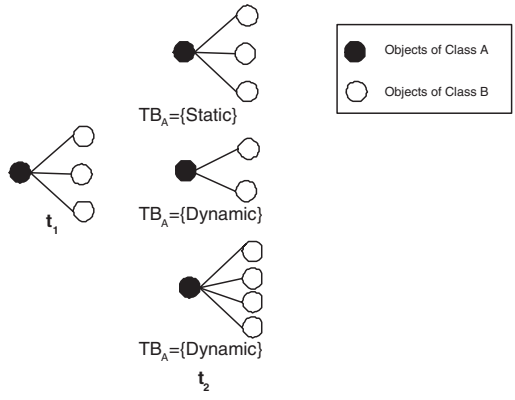


Fig. 1. A graphical example of the *temporal behaviour* dimension.

2.1 Temporal Behaviour

- `Definition`: Temporal Behaviour specifies whether an instance of a class can be dynamically connected or disconnected (creating or destroying a link[2]) with one or more instances of a related class (through an association/aggregation relationship) during its existence. The value {*Dynamic*} indicates that creating or destroying a link is possible throughout the whole life of the object. The value {*Static*} indicates that this behaviour is not possible (it is only possible during its creation process).
- `Defined over`: *association-end* (following the definition introduced on page 3-71 of the UML Standard [4]).

[2] A link is an instance of the association/aggregation relationship.

- `Nomenclature`: $\texttt{TB}_{association-end}$[3].
- `Values`: *Static* | *Dynamic.*
- `UML Related`: attribute *changeability* of the association-end: "specifies whether an instance of the Association may be modified by an instance of the class on the other end (the source end)." (page 2-22 in [4]).

Figure 1 shows an example of the *temporal behaviour* dimension. Class A is related to class B. If the value of the TB dimension defined over class[4] A is {*Dynamic*}, adding or deleting links from an object of class A to objects of class B is possible. However, if the value of the dimension is {*Static*}, adding or deleting links is not possible.

2.2 Multiplicity

- `Definition`: Multiplicity specifies the lower(`Low`)/upper(`Upp`) number of objects of a class that must/can be connected to only one object of its associated class.
- `Defined over`: *association-end.*
- `Nomenclature`: $\texttt{Low}_{association-end}$, $\texttt{Upp}_{association-end}$.
- `Values`: nonnegative integers.
- `UML Related`: attribute *multiplicity* of the association-end: "specifies the number of target instances that may be associated with a single source instance across the given Association." (page 2-23 in [4]).

2.3 Delete Propagation

- `Definition`: Delete Propagation indicates which actions must be achieved when an object is deleted (over its links and its associated objects).
  - {*Restrictive*}: if the object has links, it cannot be deleted.
  - {*Cascade*}: if the object has links, the links and the associated objects must be deleted.
  - {*Link*}: if the object has links, the links must be deleted (not the associated objects).
- `Defined over`: *association-end.*
- `Nomenclature`: $\texttt{DP}_{association-end}$.
- `Values`: *Restrictive* | *Cascade* | *Link.*
- `UML Related`: *propagation semantics.* "A consequence of these rules is that a composite implies propagation semantics; that is, some of the dynamic semantics of the whole is propagated to its parts. For example, if the whole is copied or destroyed, then the parts so are (because a part may belong to at most one composite)." (page 2-66 in [4]).

[3] If the name of the association-end (rolename) is not defined and its related class has only one relationship, we will use the name of the class as the name of the association-end.

[4] In order to simplify, throughout the paper we use the expression "defined over the class .." in place of "defined over the association-end related to the class.. "

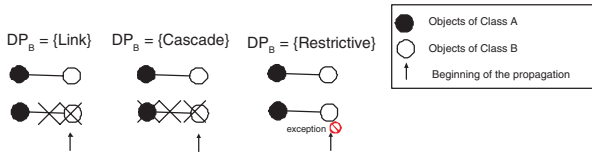


Fig. 2. A graphical example of the *delete propagation* dimension.

Figure 2 shows a graphical example of the *delete propagation* dimension. Class A is related to class B. The figure shows the different actions achieved when an object of the class B is deleted depending on the value of the dimension *delete propagation* defined over this class.

2.4 Visibility

- `Definition`: Visibility specifies whether an object can be accessed only by its associated object/s. The value {*Not Visible*} indicates that the object can be accessed only by its associated object/s. The value {*Visible*} specifies that the object can be accessed by all system objects.
- `Defined over`: *association-end*.
- `Nomenclature`: $V_{association-end}$.
- `Values`: *Visible* | *Not Visible*.
- `UML Related`: attribute *visibility* of the association-end: "specifies the visibility of the association end from the viewpoint of the classifier on the other end." (page 2-23 in [4]).

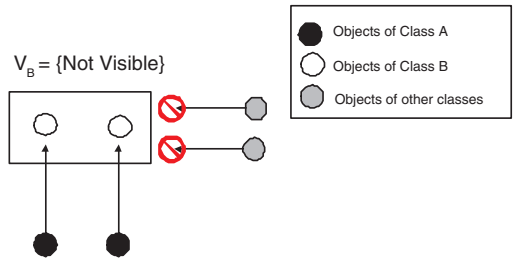


Fig. 3. A graphical example of the *visibility* dimension.

Figure 3 shows a graphical example of the *visibility* dimension. Class A is related to class B. The *visibility* dimension of class B has the value {*Not Visible*}, then objects of class B can only be accessed by their associated objects.

2.5 Identity Projection

- `Definition`: Identity Projection specifies whether an object projects its identity onto its associated object. The object keeps its own identity as its primary identity. The point is that it can also be identified by its associated object. The value {*Projected*} indicates that the object projects its identity and {*Not Projected*} indicates the contrary.
- `Defined over`: *association-end*.
- `Nomenclature`: $\texttt{IP}_{association-end}$.
- `Values`: *Projected* | *Not Projected*.
- `UML Related`: the *identity projection* of a composite: "composite [...] projects its identity onto the parts in the relationship. In other words, each part in an object model can be identified with a unique composite object. It keeps its own identity as its primary identity. The point is that it can also be identified as being part of a unique composite." (page 3-81 in [4]).

2.6 Reflexivity

- `Definition`: Reflexivity specifies whether an object can be connected to itself. The value {*Reflexive*} indicates that this is possible and {*Not Reflexive*} indicates the contrary.
- `Defined over`: relationship.
- `Nomenclature`: $\texttt{RF}_{relationship}$.
- `Values`: *Reflexive* | *Not Reflexive*.
- `UML Related`: a characteristic of aggregation and composition relationships. "[...] the instances form a directed, non-cyclic graph." (page 2-67 in [4]).

2.7 Antisymmetry

- `Definition`: Antisymmetry specifies whether an object can be connected to an object which is already connected to it. If this is possible, the value of the dimension is {*Not Antisymmetric*}. If this is not possible, the value of the dimension is {*Antisymmetric*}.
- `Defined over`: relationship.
- `Nomenclature`: $\texttt{AS}_{relationship}$.
- `Values`: *Antisymmetric* | *Not Antisymmetric*.
- `UML Related`: the *antisymmetry* property. "Both kinds of aggregations define a transitive, antisymmetric relationship; that is, the instances form a directed, non-cyclic graph. Composition instances form a strict tree (or rather a forest)." (page 2-67 in [4]).

Figure 4 shows an example of the *antisymmetry* dimension. Class A is related to class B through the `rA_B` relationship. If the *antisymmetry* dimension has the value {*Not Antisymmetric*}, then creating a link from an object of class A or B to an object of the associated class that has already a link to the former is possible. However if the value of the dimension is

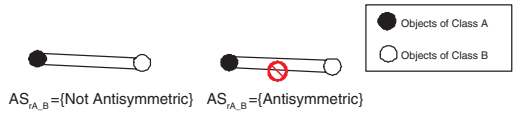


Fig. 4. A graphical example of the *antisymmetry* dimension.

{*Antisymmetric*}, the creation of a link from an object of class A or B to any object of the associated class that has already a link to the former is not possible.

The presented dimensions are used in the next section for defining a particular semantics of the UML association, aggregation and composition.

3 A Particular Semantic Interpretation

Due to the lack of a precise and clear semantics for the association/aggregation relationship, we have proposed a particular semantic interpretation for the association, aggregation and composition UML concepts in the context of the OO-Method. In order to define this semantics, we adopt only the subset of the semantics for the UML concepts that have an "unambiguous" interpretation, completing it with our own definitions. The following assertions are adopted:

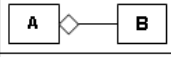
- Association: "An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship to be valid." (page 2-65 in [4]).
- Aggregation: An aggregation is "a whole/part relationship. In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations." (page in 2-66 [4]).
- Composition: A Composite aggregation is "a strong form of aggregation." (page in 2-66 [4]).

Although we adopt these assertions, we must fix the semantics of these concepts to avoid the ambiguity introduced by the UML standard. One way to define the precise semantics of these concepts is to determine the value of the dimensions that characterize each concept, allocating each concept in our multidimensional framework in a correct way.

The values of the dimensions for association, aggregation and composition concepts are presented in table 1, where the UML notation is used for representing the concepts. For each concept (columns) this table shows the value of the proposed dimensions[5]. In addition, the rows of the table show the default

[5] We use the symbol $\forall$ to show that a dimension can be set to any of its possible values.

Table 1. The fixed and default values of the dimensions for association, aggregation and composition.

Dimension / Concept	Association 	Aggregation 	Composition 
Temporal Behaviour	∀ ($TB_A = TB_B$ = Dynamic)	∀ ($TB_A = TB_B$ = Dynamic)	**TB_A = ∀ TB_B = Static** (TB_A = Dynamic)
Multiplicity	∀	∀	**$Low_A = Upp_A = 1$ Low_B = ∀, Upp_B = ∀**
Delete Propagation	∀ ($DP_A = DP_B$ = Link)	∀ ($DP_A = DP_B$ = Link)	**DP_A = Cascade DP_B = ∀** (DP_B = Link)
Visibility	∀ ($V_A = V_B$ = Visible)	∀ ($V_A = V_B$ = Visible)	**V_A = Visible V_B = Not Visible**
Identity Projection	∀ ($IP_A = IP_B$ = Not Projected)	∀ ($IP_A = IP_B$ = Not Projected)	**IP_A = Projected IP_B = Not Projected**
Reflexivity	∀ (Reflexive)	**Not Reflexive**	**Not Reflexive**
Antisymmetry	∀ (Not Antisymmetric)	**Antisymmetric**	**Antisymmetric**

values (presented in brackets) of the dimensions for each concept. The notation used for association, aggregation and composition implies the assumption of the default values.

In this table we observe that `Association` has not fixed dimensions. This is because there are no constraints in the relationship between the related classes. `Aggregation` has two fixed dimensions: *reflexivity* fixed to {*Not Reflexive*} and *antisymmetry* fixed to {*Antisymmetric*}. These values are required because in our interpretation of the whole/part relationship a given object cannot be whole and part at the same time and if an object is a part of a whole then that same whole cannot be part of its own part [6]. `Composition` has a fixed value for each dimension since: the part has to be included in one and only one composite (thus, lower and upper *multiplicity* of the composite are 1); the part cannot change its composite (thus, *temporal behaviour* of the part is {*Static*}); when a composite is deleted, its parts are deleted too (thus, *delete propagation* of the composite is {*Cascade*}); the part can only be accessed by its composite (thus, *visibility* of the part is {*Not Visible*}); a composite projects its identity onto its parts (thus, *identity projection* of the composite is {*Projected*}); as a type of aggregation, composition fulfills the constraints of aggregation (thus, *reflexivity* is {*Not Reflexive*} and *antisymmetry* is {*Antisymmetric*}).

Figure 5 shows an example of the graphical notation used for representing the values of the dimensions. The non-default values are represented at the association-end or at the relationship that they qualify using UML stereotypes following the notation introduced on pag. 3-31 [4].

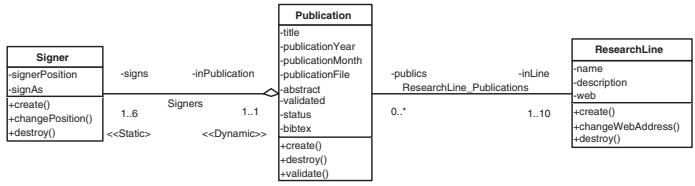


Fig. 5. Class Diagram Example

3.1 Modeling Issues

In this section we discuss the implication of the provided semantic for the conceptual modeling step. There are two issues influenced by the semantics:

- The {*Dynamic*} value of the temporal behaviour dimension defined over a class implies the existence of two operations in this class for 'connecting' and 'disconnecting' the objects of the class to objects of its related class/es. These operations are the link insertion and deletion services.
- Due to the non-orthogonality of the framework dimensions, the semantic of some dimensions constrains the possible values of others. When association/aggregation relationships are modeled, the values of some dimensions imply that other dimensions have to be set to a fixed value:
  - $\mathrm{TB}_A = \{Static\}$ implies $\mathrm{DP}_A <> \{Link\}$
  - $\mathrm{Low}_A = 1$ implies $\mathrm{DP}_A <> \{Link\}$
  - $\mathrm{IP}_A = \{Projected\}$ implies $\mathrm{Upp}_A = 1$

This knowledge can be applied to build better graphical modeling tools which include semantic checking that helps the software engineer in the construction of conceptual schemas.

Once we have proposed an interpretation for the association, aggregation and composition concepts, we are going to discuss a systematic method that obtains the software representation from an association/aggregation relationship in the conceptual schema that is characterized by the framework dimensions.

4 Code Generation Process

In this section, we define the mapping between the specification of an association/aggregation relationship in the conceptual schema and its software representation. This mapping must be unique, so that a specification of an association/aggregation relationship maps to only one software representation. In the OO-Method the mappings between the elements of the conceptual schema and their software representation are defined by applying a *code generation strategy* that is provided by the *Execution Model* [10] of the OO-Method.

We present a code generation strategy for obtaining the software representation of the association/aggregation relationships at the business tier of a three tiered architecture[6]. In order to define this strategy, we identify the software elements that must be implemented to represent an association/aggregation relationship:

- Class Structure (in terms of attributes and methods).
- Instance Creation and Destruction Services.
- Link Insertion and Deletion Services.
- *Other Services* (services that do not create or destroy instances, or insert or delete links).

In this section, we present the implementation of the elements that are influenced by the dimensions of an association/aggregation relationship. These elements are the Class Structure, the Instance Creation and Destruction Services and the Link Insertion and Deletion Services. The implementation of *Other Services* is not presented because it is independent of the dimension values.

We use the example in Figure 5 to present the implementation. This figure shows a part of an UML class diagram with three classes and two relationships. The `Publication` class is related to the `Signer` class through the *signers* relationship. This relationship is a whole/part relationship, i.e., an aggregation relationship ({*Not Reflexive*} and {*Antisymmetric*}), since a signer is part of a publication. The dimensions of the relationship have the default values for each one, except for *Temporal Behaviour* over the class `Signer`, that is set to {*Static*} (because a signer cannot add or delete a link to a publication during its existence), and *Delete Propagation* over the class `Publication`, that is set to {*Cascade*} (because when a publication is deleted, the signers associated to this publication must be deleted). Also, the `ResearchLine` class is related to the `Publication` class through the *researchline_publications* relationship. This relationship is not a whole/part relationship because it does not exist a class that is part of a composite class. Then, the relationship is an association, which does not have constraints. This relationship has the default values for all dimensions.

4.1 Class Structure

In this subsection, we present the structure of classes that implement the semantics of an association/aggregation relationship. We analyze how the values of the dimensions influence the structural definition.

Classes. Given an association/aggregation relationship between two classes A and B, we generate the implementation class C_A that implements the domain class A and the implementation class C_B that implements the domain class B. If the *visibility* dimension has the value {*Visible*} defined over one of the classes, then this class must be hidden to all classes of the system except for its associated class/es.

[6] The architectural style that the OO-Method uses to implement a system is a three-tiered architecture: presentation tier, business tier and persistence tier.

Attributes. For each implementation class we define:

- An attribute for each specified attribute in its domain class[7].
- An object-valued attribute for each association/aggregation relationship of its domain class. According to the *upper multiplicity* this attribute stores an associated object (if the value is 1) or a collection of associated objects (if the value is >1).
- If the *identity projection* dimension defined over the associated class has the value {*Projected*}, the class has as identity mechanism its identity attributes and those of its associated class.

Methods. For each implementation class we define a method for each service specified in its domain class.

Following this strategy, the class structure for the relationships in the Figure 5 is (we use the C# language to document the implementation):

```
// constants representing multiplicity values
const int UpperInLine = 10;
const int UpperSigns = 6;
const int LowInLine = 1;
const int LowSigns = 1;
...
public class Signer () {
    //...attributes
    Publication SignersObj;
    // reference to the associated publication of a signer
    //...methods
}
public class Publication () {
    //... attributes
    ResearchLine[UpperInLine] ResearchLine_PublicationsCol;
    // reference to the associated researchlines of a publication
    Signer[UpperSigns] SignersCol;
    // reference to the associated signers of a publication
    //... methods
}
public class ResearchLine () {
    //... attributes
    Publication[] ResearchLine_PublicationsCol;
    // reference to the associated publications of a researchline
    //... methods
}
```

[7] The expression "its domain class" refers to "its corresponding class in the conceptual schema".

4.2 Instance Creation and Destruction

In this subsection, we present the implementation of the instance creation and destruction methods in terms of the actions that these methods should carry out according to the values of the dimensions.

Creation Services. Each creation service, specified in a class of the conceptual schema, maps to a public method (`MCreate`) of its own implementation class. The `MCreate` method of the C_A class has to create an instance of the class A and depending on the *lower multiplicity* (if $Low_B > 0$) to connect this created instance to the necessary instances of C_B class (through an `Insertion Service` that is presented in the next subsection).

In our example, the *creation method* of the `Publication` class is implemented as follows:

```
public static Publication MCreate( ) {
    Publication p = new Publication();
    // Create a publication instance
    for (int i=1 ; i < LowSigns; i++)
        p.MInsertSigners();
        // Connect the created publication to its signer/s
    for (int i=1 ; i < LowInLine; i++)
        p.MInsertResearchLine_Publications();
        // Connect the created publication to its researchline/s
    return p;
}
```

Destruction Services. Each destruction service, specified in a class of the conceptual schema, maps to a public method (`MDestroy`) of its own implementation class. The `MDestroy` method has to destroy an instance and carry out some actions depending on the value of the *Delete Propagation* dimension:

- If $DP_A = \{Restrictive\}$: destroy the object if it does not have any link, else throw an exception.
- If $DP_A = \{Link\}$: (1) check if any associated object violates the *Lower Multiplicity* when it is > 0, (2) disconnect the associated objects to the object that is going to be destroyed and (3) destroy the object.
- If $DP_A = \{Cascade\}$: (1) destroy the associated objects and (2) destroy the object.

The *destruction method* of the `Publication` class is implemented as follows:

```
public int MDestroy ( ) {
    for (int i=0 ; i < SignersCol.Length ; i++)
        SignersCol[i].MDestroy();
        // Destroy the associated signers because DP_inPublication =
{Cascade}
```

```
        for (int i=0;i<ResearchLine_PublicationsCol.Length;i++)
            ResearchLine_PublicationsCol[i].MDisconnectResearchLine_
Publications(this);
            // Disconnect the associated researchlines to the publication
            // because DP_publics = {Link}
        MDestroyInstance( );
        // Destroy a publication instance
        return 0;
    }
```

4.3 Insertion and Deletion Services

Classes that have {Dynamic} *Temporal Behaviour*[8] must implement insertion and deletion services. Thus these classes are responsible of adding/deleting links.

Insertion Services. They create a link to an associated object. Each insertion service specified in a class of the conceptual schema maps to a public method (`MInsertRelationshipName`) in its own implementation class. The strategy for implementing the insertion method of the C_A class is the following:

1. Check if the instance of C_A that is going to be connected by the link violates the *Upper Multiplicity* when Upp_B is $<> n$.
2. `Create` or `Select` an instance of C_B class (that is going to be connected to the instance of the class C_A) depending on the value of the *Temporal Behaviour* and *Multiplicity* dimensions. If $TB_B = \{Static\}$ or $Low_A = Upp_A$ the instance is *created* (through an `Indirect Creation` method). If $TB_B = \{Dynamic\}$ and $Low_A <> Upp_A$ the instance is *selected* (through a `Selection` method). The implementation of these methods is presented throughout this section.
3. Connect the *created* or *selected* instance and the instance of the C_A class.

In our example, the `MInsertSigners` *insertion method* of the `Publication` class is implemented as follows:

```
    private int MInsertSigners( ) {
        if (SignersCol.Length < UpperSigns) {
        // (1) Check the Upper Multiplicity
            Signer sg = Signer.MCreateIndirect ();
            // (2) Create a signer to be connected to the publication
            // (Low_inPublication = Upp_inPublication and TB_signs = {Static})
            sg.MConnectSigners(this);
            MConnectSigners(sg);
            // (3) Connect the created signer(sg) to the publication (this)
            return 0;
```

[8] It implies that their instances can add or delete links during their life time.

```
        } else throw new ViolatedMultiplicityException;
    }
```

The `MInsertResearchLine_Publications` *insertion method* of the `Publication` class is implemented as follows:

```
    private int MInsertResearchLine_Publications( ) {
        if (ResearchLine_PublicationsCol.Length< UpperinLine) {
        // (1) Checks the Upper Multiplicity
            ResearchLine rl = MSelectResearchLine_Publications();
            // (2) Select a researchline to be connected to the publication
            // (Low_publics <> Upp_publics and TB_inLine = {Dynamic})
            rl.MConnectResearchLine_Publications(this);
            MConnectResearchLine_Publications(rl);
            // (3) Connect the selected researchline(rl) to the publication(this)
            return 0;
        } else throw new ViolatedMultiplicityException;
    }
```

Next, we present the strategy for implementing the auxiliary methods used in the `insertion` method, and that have been mentioned above.

Indirect Creation Method. This is a private method (`MCreateIndirect`) which creates instances of an associated class due to the insertion of a link between related objects. The `MCreateIndirect` method of the class C_A has to create an instance of the class A and depending on the value of the *lower multiplicity* (if $Low_B > 1$) to connect the created instance to the necessary instances of the C_B class.

Selection Method. This is a private method (`MSelectRelationshipName`) which selects an instance of the associated class to be connected with another instance. The strategy for implementing the selection method of the C_A class is the following:

1. Select an instance of the C_B class.
2. Check if the selected instance violates the *Upper Multiplicity* when Upp_A is $<> n$.
3. Check that the selected instance is not the same as the one to which it is going to be connected if the value of the *Reflexivity* dimension is {*Reflexive*}.
4. Check that the selected instance it is not yet connected to the one to which it is going to be connected if the value of the *Antisymmetry* dimension is {*Antisymmetric*}.

Deletion Services. They delete a link to an associated object. Each deletion service specified in a class of the conceptual schema maps to a public method (`MDeleteRelationshipName`) in its implementation class. The strategy for implementing the deletion method of the C_A class is the following:

1. Check if the associated objects violate the *Lower Multiplicity* when Low>0.
2. Disconnect the associated objects (i.e., delete the link).

The *deletion method* of the `ResearchLine` class is implemented as follows:

```
public int MDeleteResearchLine_Publication(Publication pb){
    if (pb.ResearchLine_PublicationsCol.Length>LowInLine){
    // (1) Check the Lower Multiplicity of the publication
    // The researchline Lower Multiplicity is not checked because is 0
        MDisconnectResearchLine_Publications(pb);
        pb.MDisconnectResearchLine_Publications(this);
        // (2) Disconnect the reserachline and the publication
        return 0;
    } else throw new ViolatedMultiplicityException;
}
```

Note that due to the fact that the implementation of the relationship is bidirectional (i.e., the two associated classes refer to each other) we need that the insertion and deletion methods behave as a transaction. In this way we can maintain the integrity of the references.

5 Conclusions and Further Work

In this work, we have presented a particular characterization for the association, aggregation and composition concepts introduced by the UML Standard. For the definition of this characterization we have built a multidimensional framework. This framework has been based on a study of different approaches that deal with association/aggregation relationships in OO conceptual modeling. The framework identifies the relevant properties ("dimensions") of association/aggregation relationships and allows us to give precise interpretations of the association, aggregation and composition abstractions giving specific values to the dimensions of the framework. Once we have defined the semantics of these abstractions, we propose a method for implementing the newly characterized abstractions. This method determines the mappings between the dimensions of the framework and the software elements of the Solution Space.

This approach is being incorporated to the OO-Method CASE tool, the software automatic production environment that gives support to the OO-Method.

We are developing a wizard that helps the analyst to automatically determine the kind of relationship: association, aggregation and composition by asking for modeling information through questions. These questions are oriented to determine the values of the dimensions. We are attempting to improve our proposal by identifying possible structural and behavioural patterns. Finally, in order to

improve the code generation process we are studying existing design patterns in order to select those that could be applied to the implementation of the association/aggregation relationships.

References

1. F. Civello. Roles for composite objects in object-oriented analysis and design. In ACM Press, editor, *OOPSLA'93*, pages 376–393, 1993. ISBN: 0-89791-587-9.
2. M. Dahchour. *Integrating Generic Relationships into Object Models Using Metaclasses.* PhD thesis, Dept. Computing Science and Eng., Université Catholique de Louvian, Belgium, March 1999.
3. R.C. Goldstein and V.C. Storey. Data Abstractions: Why and How? *Data and Knowledge Engineering*, 29:293–311, 1999.
4. Object Management Group. Unified modeling language specification version 1.4. Technical report, 2001.
5. B. Henderson-Sellers and F. Barbier. Black and white diamonds. In R. France and B.Rumpe, editors, In *Proceedings UML'99. The Unified Modeling Language Beyond the Standard*, pages 550–565. Springer-Verlag, 1999.
6. B. Henderson-Sellers and F. Barbier. What Is This Thing Called Aggregation? In J. Bosch R. Mitchell, A.C. Wills and B. Meyer, editors, *Proceedings of TOOLS 29*, pages 216–230, Los Alamitos, CA, USA, 1999. IEEE Computer Society.
7. J. Miller. What UML should be. *Communications of the ACM*, 45(11):67–69, November 2002.
8. J.J. Odell. Six different kinds of composition. *Journal of Object Oriented Programming (JOOP)*, 5(8):10–15, January 1994.
9. A.L. Opdahl, B. Henderson-Sellers, and F. Barbier. Ontological Analysis of Whole-Part Relationships in OO-models. *Information and Software Technology*, 43:387–399, 2001.
10. O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano. The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. *Information Systems*, 26(7):507–534, 2001.
11. A. Pirotte, E. Zimányi, and M. Dahchour. Generic Relationships in Information Modeling. Technical report, YEROSS TR-98/09 Université Catholique de Louvain, Belgium, December 1998.
12. M. Saksena, R.B. France, and M.M. Larrondo-Petrie. A Characterization of Aggregation. In Springer, editor, *Proceedings of OOIS'98*, pages 11–19. C. Rolland and G. Grosz, 1998.
13. M. Snoeck and G. Dedene. Core Modelling Concepts to Define Aggregation. *L'Objet*, 7(1), February 2001.
14. Y. Wand, V.C. Storey, and R. Weber. An Ontological Analysis of the Relationship Construct in Conceptual Modeling. *ACM Transactions on Database Systems*, 24(4):494–528, December 1999.
15. M. Winston, R. Chaffin, and D. Herrmann. A Taxonomy of Part-Whole Relations. *Cognitive Science*, 11:417–444, 1987.