Programming and Computer Software, Vol. 26, No. 4, 2000, pp. 207–215.
Original English Text Copyright © 2000 by Torres, Martin, Troyano, Toro.

Implementing Associations among Classes in an Environment of Active Databases[1]

J. Torres, O. Martin, J. A. Troyano, and M. Toro
Department of Languages and Computer Systems, University of Seville (Spain)
e-mail: jtorres(octavio)(troyano)(mtoro)@lsi.us.es

Received September 23, 1999

Abstract—The association is a native concept from relational databases, one that has been adapted to object oriented (OO) modelling. It is an interesting operator used to describe links among objects of a system, commonly included in the most popular diagram-based OO methodologies. However, those methodologies sometimes present a lack of formality that may undermine its use. In this paper we formalize the semantics of associations. Firstly, we will describe an OO model based on different kinds of constraints. Some of them will be especially useful for describing the semantics of associations. Finally, we will present some remarks about implementation by means of triggers, a new feature incorporated in databases to specify an inner active behavior.

1. INTRODUCTION

The association is a native concept from relational databases (relationships are one of the pillars of the Entity-Relationship model). This operator has been adapted to OO modelling from the very start, by some very important methods [14, 15]. Association is, together with inheritance, one of the most popular mechanisms in OO methods based on diagrams. Associations among classes are used to describe links between objects of a system. Links may be created and destroyed freely (although it is common to define several kinds of constraints to restrict this *freedom*) [16].

However, the association usually has many interpretations (an even more serious problem arises with the aggregation) [2, 7, 12]. In this work, we present a formalization of the properties of associations by means of a potent OO model. We do not intend here to give a new definition of association. Neither do we intend to substitute methods based on diagrams. These are very useful, because they facilitate communication with users and the validation of models. However, we believe that it is necessary to formalize these methods [4]. This way, our sole objective is to formalize one of the possible interpretations that can be given [3, 14, 15].

This paper is organized as follows. This introduction constitutes the first section. In the second section we will describe an OO model based mainly on the definition of constraints. In the third section, both the properties and features of associations will be described. These properties are represented in our model according to the steps described in the fourth section. In the fifth section, we will describe our main ideas in order to hold the defined semantics in an environment of active databases (like Oracle 8). Finally, in the sixth section we will extract some conclusions of our work.

2. AN OBJECT ORIENTED MODEL

Objects are the fundamental elements in any OO model. In our model, an object is characterized by a group of attributes that define its structure, a group of events that describe its behavior and some transition rules that denote the state changes of objects. Objects sharing characteristics are grouped into classes.

Each attribute has a type, defined by an *abstract data type* (ADT) or a class of objects. Values of the attributes of an object give information about its state. Attributes can be constant, variable or derived.

Each object has an *identification* that remains unchanged during its *life*. Identification should be unique for each object in the system. We consider that each object has a predefined attribute, called *oid* (object identifier) [11].

Behavior aspects of a class are described by means of events. An event describes something that happens in a moment of time. Objects interact with their *environment* by means of events, which take place through *communication channels*. These channels initially coincide with the names of events. All objects of the same class share each communication channel defined in that class. A name and several parameters that will be communicated through the events of this name define a channel. Events are very important because they are synchronization and communication elements. We define interactions between different objects with them.

Objects can be created and destroyed dynamically. All objects composing a system at a given instant interact concurrently. However, the individual behavior of

[1] This article was submitted by the authors in English.

each object is sequential. Remember that objects interact synchronously by means of events.

Other important characteristics in our model are the following:

We use an ADT library to describe the structure and functionality of objects.

Specification is carried out with different kinds of *constraints*. These constraints allow us to define three fundamental aspects of objects: (1) what values the attributes of objects can take, (2) how objects can behave in function of their state and (3) how objects can interact and with whom.

The model of interaction between objects is quite flexible. The classes of objects that should interact are defined statically, while objects of these classes that really interact are chosen dynamically. All objects fulfilling their constraints can participate.

Facilities are provided to manipulate the extension of classes (group of objects of a class that exist at a certain instant). This allows us to impose constraints on objects on multiple levels, as we will see in the next section.

2.1. Kinds of Constraints

A large diversity of constraints exists in our model [17]. According to the scope where constraints are defined, they can be of two kinds:

• *Individual constraints.* These constraints are defined in the class template. They must be fulfilled individually by all objects belonging to that class.

• *Collective constraints.* Objects of a class, considered as a collection, must satisfy these constraints, rather than individual objects.

According to the way constraints affect the objects, they can be of three kinds:

1. *Constraints on states of objects.* They allow us to define constraints on values of attributes. According to the number of states affected, these constraints can be of two kinds:

(a) *Static constraints.* They restrict the values of attributes and they should not be violated in any state. If they are fulfilled in the *current* state, they should continue being fulfilled in the *next* state. If an object does not fulfill these constraints in the initial state, it will not be created. These constraints can be defined in an individual or collective way.

(b) *Dynamic constraints.* They are bonds between two states: the current and the next. There are two kinds of dynamic constraints: (1) state changes associated with the occurrence of an event, which have the restricted form of an assignment, and (2) more generic transition constraints, which are not associated with events and act according to defined state changes. They can be defined either in an individual or a collective way.

2. *Participation constraints.* They define when an object is interested in participating in an event or when it must participate. They can be specified in two ways:

(a) *Participation permissions.* They are predicates established both on the state of an object and on the parameters of an event. If they are not fulfilled, they will prevent the object from participating in that event. They can be defined either in an individual or collective way.

(b) *Participation obligations.* Permissions uniquely allow to objects participate in an event, without assuring its participation (for example, when constraints on states are not fulfilled). If participation obligations are fulfilled, we are assured that objects will participate in that event. They can only be defined in an individual way.

3. *Interaction constraints among objects.* They define how objects interact between them (through events). Objects that should interact through an event have to:

(a) *Synchronize.* Our model is totally synchronous. It is necessary that all obliged objects participate. If some object that is obliged to participate cannot make it, then the event will not be able to happen.

(b) *Communicate.* A communication of values might take place between interacting objects. Values should fulfill all constraints imposed by those objects, both locally and globally. This way, a negotiation should be established. If more than one value is valid, the selection of the value will be non-deterministic.

Each class will have a local view of events in which it participates. By means of interactions, we unify in a single global event the different local views of that event in the participant classes.

2.2. Well-Formed Expressions

Expressions should be formed by terms that are syntactically correct. This is done by any operation defined in the library whose parameters are also syntactically correct terms or variables of the corresponding sorts (also defined in the library). Variables of these expressions can be:

• Attributes evaluated in the object itself.

• Attributes evaluated in other objects, whose identification is known. Thus, if the class cl_1 has the definition $at_1 : cl_2$, the attribute at_1 is used to identify an object of the class cl_2. Then, the expression $at_1.at_2$, being an at_2 attribute of objects of the class cl_2, is well formed. The type of this expression is the same as the type of the attribute at_2, and it denotes the value of this attribute evaluated in the object at_1.

• Parameters of events. These can only be used in specifying permissions, obligations and state changes.

Expressions for the extension can be formed in a similar way. We will consider that:

1. There exists an implicitly defined attribute that holds the set of identifications of all objects of each class. We will denote by $\overline{cl}$ the set of class $\overline{cl}$.

2. The following collection constructors can be used:

(a) $\{x_1 : c_1, \ldots, x_n : c_n \mid pred(x_1, \ldots, x_n) \bullet exp(x_1, \ldots, x_n)\}$. For each combination $x_1, \ldots, x_n$ of elements, it builds a set with values returning the expression *exp* if the predicate *pred* is true. This notation is only used if the sets $c_1, \ldots, c_n$ are finite.

(b) $[x_1 : b_1, \ldots, b_n : \mid pred(x_1, \ldots, x_n) \bullet exp(x_1, \ldots, x_n)]$. For each combination $x_1, \ldots, x_n$ of elements, it builds a bag with values returning the expression *exp*, when the predicate *pred* is true. This notation is only used if bags $b_1, \ldots, b_n$ are finite.

3. Since sets and bags are manipulated in the extension, we can also have operations like *add*, *product*, *max*, *min*, *and*, *or*, and so on. These operations are considered as generalizations of the corresponding binary operations.

2.3. Dynamics of the System

An event will be able to happen if, for each class synchronizing through this event, there is at least one interested object. If some class does not have any objects interested in participating, then that event will not be able to happen.

Constraints on states are conditions that must be fulfilled during the lifetime of objects. Since the occurrence of an event can change the value of some attributes, other established constraints should not be violated.

In order to express these ideas formally, we will define two sets. Let $cn(v)$ be an event of the system, with the channel cn and parameters v, in which the classes cl_i participate through their local views $cn_i(v_i)$, $\forall i \in \{1..n\}$. We define:

• $\mathcal{P}_{cn_i(v_i)}$ to denote the set of objects of the class cl_i that can participate through the local view $cn_i(v_i)$ of the event. This set is composed of those objects of cl_i that fulfill their permissions on the local view $cn_i(v_i)$ and its constraints on states are not violated (at any level). Then, the set of objects that can participate through all local views of $cn(v)$ will be

$$\mathcal{P}_{cn(v)} = \bigcup_{i \in \{1\ldots n\}} \mathcal{P}_{cn_i(v_i)}.$$

• $\mathcal{O}_{cn_i(v_i)}$ to denote the set of objects of cl_i that must participate through the local view $cn_i(v_i)$ of the event. This set is composed of those objects of cl_i that fulfill its obligations of participating in $cn_i(v_i)$. The set of objects that must participate through all local views of $cn(v)$ will be:

$$\mathcal{O}_{cn(v)} = \bigcup_{i \in \{1\ldots n\}} \mathcal{O}_{cn_i(v_i)}.$$

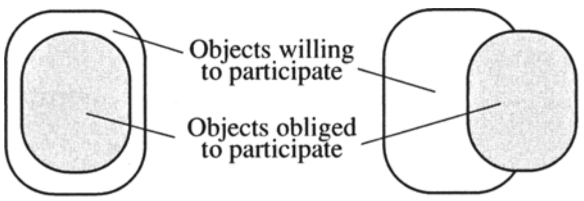
Objects willing to participate
Objects obliged to participate

Fig. 1. When can happen an event?

An event will be able to happen if $\mathcal{O}_{cn(v)} \subseteq \mathcal{P}_{cn(v)}$ (Fig. 1) and $\mathcal{P}_{cn_i(v_i)} \neq \emptyset$, $\forall i \in \{1..n\}$. Finally, if the event $cn(v)$ happens, all objects $o \in \mathcal{P}_{cn(v)}$ will carry out the state changes associated to this event.

In short, in our model, several classes can participate in an event and for each class all those objects fulfilling their constraints. So, we have a more flexible communication model than the traditional *client-server* approach.

3. ASSOCIATIONS AMONG CLASSES

In OO systems, the state is structured on different levels. This way, the state of an object is defined by the values of its attributes at a given moment. The state of the system, in principle, is defined by the state of all object states composing it at a given moment.

However, the state of a system cannot always be described in this way. Such a state should also contain *links* between objects. These links are specified by means of associations; i.e., links are instances of associations.

3.1. Characteristics of Associations

According to the number of classes involved, associations can be of three types: *binary*, if they are defined between objects of two different classes, *unary*, if they are defined between objects of the same class and *complex*, if they are defined between objects of three or more classes. Henceforth, we will not consider the last one because it can become a set of binary associations.

An association is defined by (1) a *name* (2) the *role* played by objects of a class with regard to the other class, and (3) the *multiplicity* of each role. The multiplicity indicates how many objects of a class can be related with an object of the other class of the association.

Associations are commonly represented as continuous lines between the participant classes in the relationship, as shown in Fig. 2 (in UML notation [3]), where the name of the association is omitted for reasons of clarity. At each endpoint of the line the role and the multiplicity of the nearest class is indicated. This denotes that each object of $Class_1$ can be related with $multiplicity_2$ objects of $Class_2$. On the other hand, each

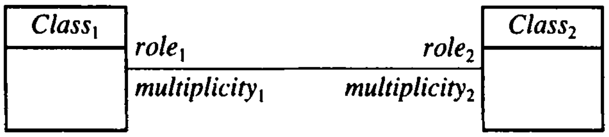

Fig. 2. UML representation of a binary association.

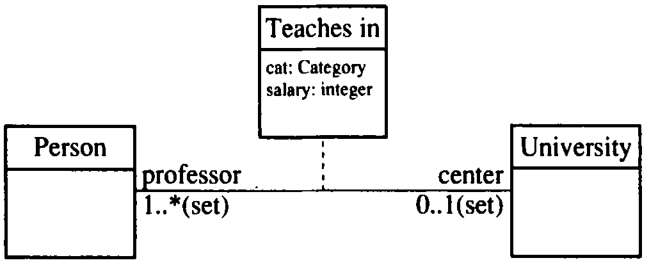

Fig. 3. Association with attributes.

object of $Class_2$ can be related with the $multiplicity_1$ objects of $Class_1$.

Multiplicity is defined by means of a range notation (*inf..sup*). The lower limit specifies the minimum number of objects linked with the given one. According to this number, an association can be mandatory (positive number) or optional (0). The upper limit specifies the maximum number of objects linked with the given one. If the lower and upper limits coincide, a unique number will be indicated. An asterisk (*) denotes a non-existing upper limit [2, 3, 7].

In Fig. 2, $role_1$ denotes the role that objects of $Class_1$ play with regard to the $Class_2$. It is like a function that, given an object of $Class_2$, returns the associated object or collection of objects of $Class_1$. On the other hand, $role_2$ has a similar meaning. Roles can be organized in a set (unordered collection of objects of the same class, without duplicates) or bag (unordered collection of the same class of objects, with duplicates).

We can also define attributes for associations. These attributes will take value when objects are associated, but they do not belong to those objects. Similarly, events and transitions can be added like in classes. Therefore, we have a homogeneous treatment for classes and associations. Figure 3 shows the association between a university and people that are professors of this university. Each professor belongs to a *category* and has a *salary*. These attributes are not common to the rest of the people. This way, in the association *teaches in* there will be a link with these attributes for each professor working at the university.

Another interesting property that can be considered in associations is the exclusivity. By default, we consider that participation of a class in an association is not exclusive. If a class has more than one association, and in some of them its participation is exclusive, objects with links in the exclusive association cannot have links in others. This way, in the previous example we can also define the association *studies in* between *University* and *Person*. Now, for instance, we can define a constraint indicating that a professor cannot be a student, or that a person cannot be in both associations.

3.2. Dynamics of Associations

In the previous section we have defined static aspects of associations. In this section we will express dynamic aspects of associations; i.e., how links between two objects are established and eliminated. We have to remember that objects of associated classes are obliged to certain things. They are not able to *work* independently.

As we have said, links can be created and destroyed. Thus, in association shown in Fig. 3a person can leave his job as professor in a university (whether he finishes his contract or for another reason). So it is necessary to remove the corresponding link. However, we will not be able to destroy a link if it implies violating some of the defined constraints (for example, about the multiplicity).

When an associated object disappears, its links should also be deleted. A link will not be able to exist if the object it connects does not exist. So, in the example in Fig. 3, if a university is eliminated, all links to professors that teach in that university will also be eliminated.

4. FORMALIZING ASSOCIATIONS AMONG CLASSES

There are two common approaches that are adopted when associations are represented in languages that do not have a corresponding high-level mechanism [7–9, 12]:

1st approach. Representing associations by means of attributes in associated classes. It is the most basic form. The main problem is that attributed associations cannot be represented, but can only represent roles.

2nd approach. Representing associations by means of classes and a set of constraints to hold their properties.

We will follow the second, more general approach. We will mainly make use of simple classes, collective constraints and interaction constraints of our model.

Let *as* be an association between the classes cl_1 and cl_2 taking the roles $role_1$ and $role_2$, respectively. Things to do to hold constraints imposed by that association are the following:

1. The association will be represented by a class that will initially have all its characteristics. Each object of this class will represent a link between two objects of associated classes.

2. It is necessary to add to this class the following concepts:

(a) Two constant attributes of the associated class types to represent the identifications of linked objects. We will denote these attributes by the name of the corresponding roles of the associated classes.

(b) Two channels for the collective destruction of links, so that when we destroy an object of an associated class, its links also be destroyed. It is also necessary to add both permissions and obligations so that all implied links participate (and only those links). We denoted by cn_{ds_i} the channel of collective destruction of the class cl_i, $\forall i \in \{1..2\}$; its permissions and obligations will be the following:

$$cn_{ds_i}(sid, id_i)$$
$$permissions\ oid \in sid$$
$$obligations\ \ oid \in sid$$

The channel cn_{ds_i} will have two parameters *sid* and id_i where *sid* is the set of links of that object of cl_i whose identification is id_i.

(c) It is also necessary to add permissions to this event in the extension in order to calculate which are the implied links. The set *sid* that id_i has with objects of the other class of the association, is calculated as follows:

$$cn_{ds_i}(sid, id_i)$$
$$permissions\ sid = \{y : \overline{as} | y.role_i = id_i \bullet y\}$$

(d) State changes for events of the collective destruction of links, in order to eliminate these links from the extension:

$$cn_{ds_i}(sid, id_i)$$
$$state\ changes\ \overline{as}' = \overline{as} - sid$$

where $\overline{as}'$ denotes the value of the attribute $\overline{as}'$ in the state following the occurrence of an event.

(e) Constraints on the extension to hold constraints imposed by roles defined in the association. Firstly, it is calculated what objects of a class are linked with a given object of the other class. For example, for the class cl_1 we have:

$$link_1(\overline{as}, id_i) = [y : \overline{as} | y.role_1 = id_1 \bullet y.role_2],$$

where id_1 is the identification of an object of cl_1.

Afterwards it is necessary to verify that constraints on the corresponding roles are fulfilled, i.e., constraints on both the number of objects and the kind of organization. Therefore, for the class cl_1, the following predicates must be calculated:

$$pred_1(\overline{as}, id_1) = let\ m = link_1(\overline{as}, id_1)\ in$$
$$in\_range_2(\#m)\ and\ pr_2(m)$$
$$end\ let$$

$$in\_range_2(n) = (n >= inf_2) \text{ and } (n <= sip_2)$$

$$pr_2(m) = \begin{cases} is\_set(m)\ if\ org_2\ is\ set \\ true\ if\ org_2\ is\ not\ set, \end{cases}$$

where the predicate *in_range* controls that the number of objects linked with the given one is in the correct range. If the upper limit is an *, it is not necessary to specify the condition and ($\#m <= sup_2$). On the other hand, the predicate *pr* verifies that the organization is the correct one. The necessary predicates for cl_2 are obtained in a symmetrical way.

Finally, it is necessary to verify that all objects of the associated classes fulfill the previous predicates. So we will define the following static constraint on the extension:

$$and([id_i : \overline{cl_i} \bullet pred_i(\overline{as}, id_i)]);\ \forall i \in \{1..2\}.$$

Such and operation is the and operation on Booleans extended to operate with a Boolean bag.

3. We should extend the interaction constraints so that whenever an object is destroyed, its links are also destroyed. So, for each interaction where the channel of destruction of cl_i appears, it will be necessary to include the channel cn_{ds_i}.

4. All links are among existing objects. Therefore, it is necessary to add collective static constraints. We should:

(a) Define, in the extension of the class *as*, a derived attribute for each class in the association. This attribute will be a bag referring to those objects that have a link with an object of the other class:

$$ate_{cl_i} = [id : \overline{as} \bullet id.role_i];\ \forall i \in \{1..2\}.$$

(b) In order to verify that those objects really belong to the extension of cl_i, we will add the following global constraint:

$$btos(\overline{as}.ate_{cl_i})\ inc\ \overline{cl_i};\ \forall i \in \{1..2\},$$

where the *btos* operation, given a bag of elements, returns a set without duplicates.

(c) If the participation of a class in an association is defined as exclusive, then objects in that association cannot have links in other associations. We will have to add more collective constraints. This way, if the class *cl* has defined the associations as_j with the classes cl_j, $\forall j \in \{1..m\}$, and exclusive participation in as_i, then we will have:

$$(\overline{as_i}.ate_{cl} \cap \overline{as_j}.ate_{cl}) = empty;\ \forall j \in \{1..m\}, j \neq i,$$

where ate_{cl} is the derived attribute defined in the association as_j, $\forall j \in \{1..m\}$, in order to hold identifications of objects of the class *cl* with some link in the association.

5. IMPLEMENTATION WITH ACTIVE DATABASES

Traditionally, binary associations have been maintained in relational databases by means of referential integrity. However, when associations are a bit more complex, we will need to make use of another method to maintain them. The design of active rules allows us to define procedural actions to be carried out for repairing an integrity violation, although it loses the declarative advantage of being able to specify the constraints.

In the following sections, we will describe the main points of interest necessary to implement previously defined semantics by means of active databases [6], a new research area that increases the functionality of traditional databases with active rules, providing an efficient and uniform mechanism for developing some tasks within the kernel of a database.

5.1. An Overview of Active Databases

Most of database systems are passive; i.e., data can be inserted, modified or deleted as a result of requests from either users or applications. A recent research area aims to extend the functionality of database systems by including certain type of active behavior in the database. In this way, database systems can execute some processes automatically in response to the occurrence or satisfaction of events or conditions [5, 18].

Other terms, synonyms for active rules, are production rules, event-condition-action or *ECA* rules, triggers, monitors, and so on. Although several implementations of active rules exist in database systems, there are three main components:

Event is the direct cause of the active rule to be triggered.

Condition must be satisfied so that the active rule can be triggered.

Action is the procedure to be executed when the corresponding event occurs and the condition is satisfied.

Several areas exist where active rule sets can be used with the purpose of improving the efficiency of the system. The most important activities are:

• *Internal tasks*, such as maintaining all kinds of constraints and derived data. One of them is the maintenance related to associations, which is the objective of our work.

• *Extended tasks*, such as replication, versioning and workflow management.

• *External tasks*, such as the business rules of any application.

These rules can be shared by all applications accessing the database, guaranteeing knowledge independence because the part of behavior that is traditionally accomplished by applications is moved into database systems.

Unfortunately, if the design of active rules was not appropriate, we could be in trouble because of their collective behavior, interactions and mutual influences, mainly due to the ability of rules to trigger each other. To ensure the global correctness of active rules at large, we should design an active rule set that accomplishes the termination property. Sometimes, other properties also must be taken into account, such as confluence and observable determinism [1]. Following, we define these terms:

• *Termination.* A set of active rules is said to possess the termination property when the rule processing triggered by every user-defined transaction is eventually terminated, producing a final state.

• *Confluence.* A set of active rules is said to guarantee the confluence property when such processing eventually terminates, and always produces an unique final state that is independent of the execution order of the rules.

• *Observable determinism.* A set of active rules is said to guarantee an observable determinism when, in addition to confluence, for each user-defined transaction, all visible actions performed by the rules are the same.

Another important characteristic of an active rule is the time when its action will be executed with respect to the event time and in relation to the current transaction. An active rule is said to be *immediate* if the action is executed immediately after the event occurs (if condition were satisfied), and it is said to be *deferred* if the action is executed at the end of current transaction.

The latest versions of DBMSs, both relational and object-relational, include triggers (which is the term normally used in practice). Unfortunately, a problem related to their implementation is the fact that triggers implemented in commercial database systems (such as Oracle, DB2, Sybase, Interbase, among others) are not powerful enough. This is the case because no complete standartization about triggers in SQL exists, and none of them offers deferred triggers at all. The current SQL3 specification of triggers is rather long and difficult to understand, and differences between proposals considered by standardization committees (both ANSI and ISO) [10] are an additional source of confusion.

There are other problems associated with rules. One of them is known as *mutating tables*, and it is related to problems that may arise because of transaction management: we can neither modify nor read rows of tables already updated during the transaction. Another point of interest is the incompatibility between declarative referential integrity and triggers. Although most database systems offer facilities, when we have to implement more complex relationships, we need to use triggers and such facilities should not be used.

[5] offers a partial solution to those problems. *Metatriggering* consists in a mapping from every active rule to a concrete stored procedure that codes both its condition and its action. When an event is triggered, a flag is updated in a temporal table for each active rule that is triggered by that event, and immediate rules are processed afterwards. At the end of the transaction, all deferred rules will be processed. When implementing it, we have realized some extensions to the method, so that any number of the same trigger instances could be

processed, and also allowing pass of object identifiers being affected by operations from triggers to stored procedures (all by means of time stamps).

5.2. Active Rules for Associations

As mentioned above, every class can be implemented by means of a table in a relational database, where each object will be stored in a row. Associations, like any class, can also be implemented by means of a table, where links are stored in such a table by means of references to each participant object.

Channels of events can be implemented by means of triggers reacting to the creation and deletion of objects. So, efficiency is improved because the explicit treatment of channels of events, otherwise very expensive, is avoided.

Although creation is implicitly formalized in semantics, the execution of a trigger after the creation of every object is necessary to test the constraints, such as referential integrity, multiplicity and exclusivity.

After an object is deleted, a trigger will be executed, and all links where that object participates should be deleted. Like every rule, the deletion rule has three components: event, condition and action. The former is easy and has a direct script. The condition of the rule is more complex: it is necessary to verify that, for each link where an object participates, constraints of corresponding roles are fulfilled after that object has been deleted. The action will be to propagate deletion to all links for every link where the object participates.

Creation and deletion of a link are similar. After a new link is created, participant objects must exist, and multiplicity and exclusivity must be satisfied immediately afterwards. After an existing link is deleted, the unique constraint that should be satisfied is the multiplicity.

5.3. An Example of a Trigger Set

In this section we show, using Oracle syntax [13], definition of the example in Fig. 3. Now, *Person*, *University*, and *TeachesIn* classes will be tables storing objects and links of the respective classes. No additional properties are visible. Definitions of tables are as follows:

```
CREATE TABLE Person
   (Oid INTEGER PRIMARY KEY);
CREATE TABLE University
   (Oid INTEGER PRIMARY KEY);
CREATE TABLE TeachesIn
   (Professor INTEGER, Center INTEGER, PRIMARY KEY(Professor, Center));
```

Triggers defined for the creation and deletion of *Person* objects are as follows:

```
CREATE TRIGGER CreatePerson
AFTER INSERT ON Person
FOR EACH ROW
-- WITH DEFERRED EXECUTION
BEGIN
   IF NOT (RefIntegrityACP(New.Oid)
        AND MultiplicityACP(New.Oid)
        AND ExclusivityACP(New.Oid)) THEN
     RAISE_APPLICATION_ERROR(-20000, 'Constraints are not fulfilled');
   END IF;
END;

CREATE TRIGGER DeletePerson
AFTER DELETE ON Person
FOR EACH ROW
-- WITH DEFERRED EXECUTION
BEGIN
   IF NOT MultiplicityADP(Old.Oid) THEN
     RAISE_APPLICATION_ERROR(-20000, 'Constraints are not fulfilled');
   END IF;
   DELETE FROM TeachesIn WHERE Professor = Old.Oid;
END;
```

Triggers for the *University* class are similar. Triggers *CreatePerson* and *DeletePerson* will raise an exception when any of the predicates returns *FALSE.* For brevity, stored procedures implementing the predicates are not shown. Following is a short description:

RefIntegrityACP receives the identifier of a new object and returns *TRUE* if the referential integrity is satisfied for this object, or *FALSE* otherwise.

MultiplicityACP receives the identifier of a new object and returns *TRUE* if the multiplicity constraints are satisfied for the associations in which this object participates, or *FALSE* otherwise.

ExclusivityACP receives the identifier of a new object and returns *TRUE* if the exclusivity constraints are satisfied for the aggregations in which an object participates, or *FALSE* otherwise.

MultiplicityADP receives the identifier of an old object and returns *TRUE* if the multiplicity constraints are satisfied for the associations in which this object participates after this object has been deleted, or *FALSE* otherwise.

Finally, triggers for the creation and deletion of *TeachesIn* links are similar. A link can be created if the referential integrity, multiplicity and exclusivity constraints are satisfied by the participant objects. Otherwise, an exception will be raised and no link will be created. On the other hand, a link can be deleted if the multiplicity constraints are satisfied after such an event occurs.

5.4. Final Remarks on the Implementation

Each of these rules should be deferred execution, allowing intermediate states that could temporally violate the constraints, but that are necessary when a relationship is created or deleted. As an example, we could create an object and all links where such object participates afterwards. If the rules are not deferred when the object is created, an exception could be raised because some of the constraints were not satisfied, despite links created following the creation of such object.

Many other combinations can be found, meaning that all related operations should be grouped into transactions. On commit, when these rules are processed, if any predicate is not true, then the operation must be canceled. This is implemented by raising a user-exception that rollbacks the current transaction, and therefore, undoing all those changes that were pending of being committed. As mentioned above, some additional techniques, such as meta-triggering, will be necessary to implement them because of the faults drawbacks inherent in current active databases.

Generally, the use of active rules has a better performance, because many activities, otherwise executed by an application, are running within a database, and so communications between applications and databases have a major efficiency.

Unfortunately, this set of rules guarantees termination, but not confluence nor observable determinism. So, non-determinism could arise and these situations are not recommended in many systems. Solutions could lie in applying several techniques to avoid it, such as assigning priorities and redesigning rule sets, among others.

6. CONCLUSIONS

We have presented in this paper a formalization of semantics for one of the more common operators in the OO conceptual modelling, associations among classes of objects.

First, we have presented the most important characteristics in our OO model. A specification is carried out by means of constraints of different kinds that can be imposed at three different levels: object, class and global.

Afterwards we have defined some properties of associations of classes, inspired fundamentally in the more popular OO methodologies. Next, we have defined the semantics of associations by means of classes and constraints allowed in our model. This way, to change properties of any association, it will simply be necessary to add new constraints or to eliminate some of them.

Finally, we have shown some remarks on our implementation by means of active databases. We are actually working hard to automatically generate a set of active rules from the formalized definitions of a system. Furthermore, we have planned to study other defined relationships among objects, together with their implementation, such as aggregations or inheritance.

ACKNOWLEDGMENTS

This work was supported by the Inter-ministerial Commission of Science and Technology (CICYT) of Spain, project no. TIC97-0593-C05-03.

REFERENCES

1. Aiken, A., Widom, J., and Hellerstein, J.M., Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1992, pp. 59–68.
2. Bock, C. and Odell, J., A more Complete Model of Relations and Their Implementation., *J. Object-Oriented Programming*, 1997, June, pp. 38–47.
3. Booch, G., Jacobson, I., and Rumbaugh, J., *The Unified Modeling Language User Guide*, Addison–Wesley, 1999.
4. Bourdeau, R.H. and Cheng, B.H.C., A Formal Semantic for Object Model Diagrams, *IEEE Trans. Software Eng.*, 1995, October.
5. Ceri, S. and Fraternali, P., *Designing Database Applications with Object Rules*, Addison-Wesley, 1997.

6. Dittrich, K., Gatziu, S., and Geppert, A., The Active Database Management System Manifesto: A Rulebase of a ADBMS Features, *J. SIGMOD Record*, 1996, vol. 25, no. 3, pp. 40–49.
7. Ehlmann, B.K. and Riccardi, G.A., An Integrated and Enhanced Methodology for Modeling and Implementing Object Relationships, *J. Object-Oriented Programming*, 1997, May, pp. 47–55.
8. Graham, I., Bischof, J., and Henderson-Sellers, B., Associations Considered a Bad Thing, *J. Object-Oriented Programming*, 1997, February, pp. 41–48.
9. Hammond, J., Producing Z Specifications from Object-Oriented Analysis, in *Z User Workshop 1994. Workshops in Computing*, Nicholls, J.E., Ed., Springer, 1995, pp. 316–336.
10. *ISO-ANSI. Database Language SQL3*, working draft, 1994.
11. Khoshafian, S.N. and Copeland, C.P., Object Identity, *Object-Oriented Programming Systems, Languages and Applications*, SIGPLAN Notices, 1986, vol. 22, no. 12, pp. 406–416.
12. Lano, K., *Formal Object-Oriented Development*, Springer, 1995.
13. *PL/SQL User's Guide and Reference, Release 8.0*, Oracle Corporation, 1997.
14. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
15. Shlaer, S. and Mellor, S., *Object Lifecycles: Modelling the World in States*, Yourdon Press Computing Series, 1992.
16. Torres, J., Troyano, J.A., and Toro, M., Operators of Association and Aggregation in an Object Oriented Specification Language, *II Workshop on Computer Science*, Granada (Spain), 1996, pp. 11–21.
17. Torres, J., Object Oriented Specifications Based on Constraints, *PhD Thesis*, Department of Languages and Computer Systems, University of Seville, 1997.
18. Widom, J. and Ceri, S., *Active Database Systems: Triggers and Rules for Advanced Dababase Processing*, San Francisco: Kaufmann, 1996.