The Operational Semantics of User-Defined Relationships in Object Oriented Database Systems

Oscar Díaz

Departamento de Lenguajes y Sistemas Informáticos

Universidad del País Vasco / Euskal Herriko Unibertsitatea

Apd. 649, 20080 San Sebastián, Spain

e-mail: jipdigao@si.ehu.es

Fax: +34 43 219306

Tfno.: +34 43 218000

Abstract

In semantic data models, abstract relationship (e.g. generalization, aggregation, etc) semantics is defined specifying how insertion, deletion and modification operations made at a higher abstraction level can affect the object abstracted and vice versa. This semantics, also know as *structural constraints*, is expressed through the so-called *update rules*. This perspective is somehow fade out in most object-oriented systems where user-defined relationships are supported as simple pointers and their semantics are embedded, distributed and replicated within the operations accessing these pointers. This paper inherits and extends the treatment of relationships found in semantic data models to behavioural object-oriented models by presenting an approach to uniformly capture the update rules for *user-defined relationships*. The stress is *not* on supporting relationships as first-class objects but on describing their update rules (or operational semantics) through a set of constructors namely, reaction, anticipation, delegation and exception. The approach has been born out by an implementation in an active object-oriented database system.

Keywords: Conceptual Modeling, Object Orientation, Active Databases

1 Introduction

One of the rationales behind semantic data models (SDMs) is to provide more powerful abstraction which facilitate a direct mapping with the concepts of the Universe of Discourse (UoD). Generalization, aggregation, classification and association are some of the best known abstractions used in SDMs [23]. For each abstract relationship a clear semantics must be defined specifying how insertion, deletion and modification operations made at a higher abstraction level (e.g. *person*) can affect the object abstracted (e.g. *student, lecturer*) and vice versa. This semantics, also know as *structural constraints*, is expressed through the so-called *update rules*. As an example, in [6] a set of update rules to support the structural constraints of the SDM data model are given. For instance the following rule is provided: "Let Tc be a subclass of Tp with derivation 'specified by the user'. Inserting an instance A to Tc will cause the same instance to be inserted to Tp if it is not there already. Deleting an instance B from Tp will cause deletion from Tc if B is also an instance of Tc. Insertion to Tp or deletion from Tc will not be propagated (by default)". This rule specifies the insertion and deletion semantics of the generalization abstraction. Similar rules are provided for other subclass constructors in the SDM data model.

It is worth noticing that in relational databases, tables can also materialise hierarchies. However, the difference is that here the semantics of the generalization relationship is in the *user programs* which have to enforce the corresponding structural constraints, whereas in a SDM these relationships are provided as *primitives* and the user has only to *specify* the hierarchy, leaving the semantic maintenance to the system itself. That is, the update rules are embedded, distributed and replicated among the user programs accessing the database.

This situation is slightly improved in behavioural object-oriented (BOO) data models [1] where the consideration of the behavioural features of an entity allows to concentrate the support for update rules within the operations (i.e. methods) attached to the class rather than these rules being replicated among the user programs. Relationships are represented as pointers between the affected classes, and most of their semantics are embedded within the operations accessing these pointers.

Nevertheless, update rules are still distributed and implicit among different operations. Complex objects are a case in point. The relationship semantics between a *whole* and its *parts* (e.g. the existence dependency constraint) can be supported by embedding within the operations (e.g. the *delete* method attached to the *whole*) the corresponding propagation (e.g. to propagate the deletion to the *parts*). Besides jeopardizing method modularity (e.g. what if the *delete* method is overridden), this approach still causes the relationship

[1] Here the terminology introduced in [12] is used to distinguish among the different approaches to object orientation.

semantics to be replicated for each class of complex objects. Furthermore, it is not only a programming issue but a philosophical matter as well. Relationships play an important role in the UoD and thus, they should be regarded as proper constructors rather than being implicit in other entities. This is the perspective of the SDM community that somehow fade out in BOO models.

However, the availability of powerful relationship is seen as a main requirement to cope with the variety and complexity of the UoD to which OODBs are being applied [21]. Although some systems begin to incorporate different kinds of relationships (e.g. *part_of, version_of* [18]), it is impossible to foresee the requirement of all possible applications. Hence, a general mechanism should be provided to specify user-defined relationships. Several authors support this idea and recently different systems have been enlarged to explicitly incorporate relationships as primitives [25, 14, 9]. Such approach enhances analysis, specification and understanding of the database.

Most of the approaches characterize relationships by specifying the degree, the cardinality, the inverse-link constraint, the participant classes and the attributes of the relationship. However this is not enough. In the same way that abstract relationships in SDMs come with an operational semantics built-in (i.e. the update rules supporting the relationship semantics), a mechanism should be provided to the designer of user-defined relationships for describing how operations made on an object can affect the related objects.

Since BOO systems model both the structural and behavioural features of the UoD, updates rules can be defined not only on insertions, deletions and updates as in SDMs, but also on any other operation described in the object interface (e.g. *display, move*). For example, as part of the *working_in* relationship, the billing of the business expenses to the *employee* (the relationship domain) can be delegated to the related *department* (the relationship range). It is worth pointing out that such propagations is part of the relationship semantics, and not of the operation definition. The expenses are delayed since the affected employee is *working_in* related with a *department*. Otherwise, his/her expenses would not be delegated!

This leads to a new mechanism for sharing behaviour. However, unlike previous approaches where sharing is defined at the class level, now *relationship-based sharing* can be specified. Hence, the behaviour of an object comes not only from the class to which it belongs but also from the classes to which it is related. This is after all quite realistic!

This paper inherits and extends the treatment of relationships found in SDMs to BOO models by presenting an approach to uniformly capture the update rules for user-defined relationships. Most previous approaches focuses in abstract relationships (e.g. *generalization, association*) and update rules refer to simple operations. Here, a set of constructors are proposed to declaratively define the operational semantics of user-defined relationships. The approach has been born out by an implementation in ADAM, and object-oriented

DBMS implemented in Prolog.

One significant problem when supporting relationship semantics is its inherent non-local nature since it can affect two objects: the domain object and the range object. This could lead to violate the modularity of the affected objects as well as for the class designer to be aware of the relationship in which the class may participate. The solution is to use a mechanism which allows to monitor the related objects without affecting them. *Event-condition-action rules* have been proposed for such tasks, and in the context of this work, they are seen as the 'assembler' language which is automatically obtained from the higher-level update rules given by the designer.

This paper looks at the primitives which are used to specify the operational semantics of user-defined relationships. This is the topic of section 2. Section 3 outlines how these ideas are born out in ADAM, an object-oriented database system. The underlying implementation mechanism using event-condition-action rules is shown in section 4. Section 5 compares related works. Section 6 reviews the paper and presents some conclusions.

2 Update rules for user-defined relationships

SDMs specify how abstract relationships behave regarding insertion, deletion and modification operations on the related abstract objects. Since, in OODBs, objects are described not only by their attributive features but also by their behaviour (i.e. methods), a similar question to the one arising in SDMs can be posed but now in the context of user-defined relationships and *ground* objects (i.e. instances): how does a user-defined relationship behave in response to operations performed on its participant objects?

Furthermore, in [26] it is shown how the propagation of an operation through a network of objects is often determined by the nature of the relationships between the objects, rather than the actual operation.

For instance, in terms of the standard *copy* operation defined in [17], two extremes are supported in the propagation of *copy*: the *shallow-copy* whereby the attributes of the original object are moved directly to the copy, with no recursive copying of related objects, and the *deep-copy* where the scalar attributes of the original object are moved directly to the copy, and all object-valued attributes are assigned deep copies of their original values. As pointed out by [26], these extremes of behaviour may not always represent the required behaviour. For example, in taking a copy of a *whole* it may be desirable to also copy the *parts*, but it is not likely that the *company* which makes the part should be copied as a side-effect of the copying of the part. In this case, the copy operation should be propagated over the *part_of* but not over the *made_by* relationship. That is, the *copy* operation is propagated along the *part_of* relationship. Thus, propagation is a property of the *part_of* relationship rather than a feature of the *copy* operation.

Such propagation along relationships, involves three aspects namely:

- *the triggering operation (TO)*
- *the induced operation (IO)*
- *the coupling mode* which specifies when is the induced operation executed relative to the triggering operation

As an example, consider a census database where the *marriage* relationship is defined. Assume that the class *person* has a method *house_removal* attached to it. The designer may be interested in modeling the situation where the house removal of a married *person* involves the house removal of his/her partner, i.e. when the message *house_removal* is sent to a *person*, this message is propagated to his/her partner *through the marriage relationship.* Here both the triggering and induced operations are the same, the *house_removal* method, whereas the coupling mode can be to first execute this method for the initial receiver and later, to invoke *house_removal* for his/her partner.

The coupling mode allows for the following alternatives:

- *reaction* (or *after* mode) whereby an operation TO once executed in an object O can induce an operation IO on a related object O'. The induced operation occurs after the triggering operation. This option is the one commonly found in SDMs where an operation, if properly executed, can propagate the same action to other entities. As an example consider the *part_of* relationship between *parts* and *wholes*. Deleting a *whole* can lead to delete all its *parts*, i.e. the delete operation is propagated from the whole to the parts. But first, it has to be assured than the *whole* can be deleted.

- *anticipation* (or *before* mode) whereby an operation TO on an object O delays its execution till another operation IO is successfully performed on a related object O'. For instance, consider a relationship *register_in* between *citizens* (objects O) and *councils* (objects O'). Suppose the situation where the renewal of the driving license (i.e. TO) previously requires to satisfy all the unpaid fines (i.e. IO). Such situation cannot be properly represented by a reaction mode since the payment of the fines must be previous to the renewal of the license. Neither can it be reflected by inducing the renewal once the fines are paid since this could not be the intention of the driver. Rather it is once the driver decides to renew the license that the fines have to be previously paid.

- *delegation* (or *instead_of* mode) whereby an operation TO on an object O' is performed instead of being answered by the initial receiver O as a result of its relationship with O'. As an example, consider the relationship *working_with* between *employees* and *companies*. The designer can be interested in modeling the situation where the

expenses of an employee (the initial receiver) are delegated to his/her company instead of being paid by himself. However, such delegation should be transparent to the sender (i.e. the collector for instance, the VISA company) which just ask for the payment to who got the service (i.e. the employee).

- *exception* (or *failure* mode) whereby the rising of an exception in TO when sent to O is handler by invoking TO' on a related object O'. The success of TO is conditioned to the success of TO'. For instance, consider the *marriage* relationship and assume that the class *person* has a method *invoice* attached to it. If an exception is raised when a *person* is invoiced (e.g. because a transaction failure) an attempt can be made to overcome the situation by sending the invoice to his/her partner, provided authorization is given. Unlike delegation, an exception propagation occurs only if the first message fails to fulfil its goal. This approach can be seen as a relationship-based mechanism to the n-version programming philosophy to cope with failures, where different programs are available to solve the same goal. A failure in a program leads to attempt to achieve the same goal by other alternative program.

  An exception in a method's execution is the failure of any one of the actions performed by that execution. The difference between exception and failure comes from the possibility of the calling routine (i.e. the one raising the exception) to have an alternative strategy to overcome the failure (i.e. the so-called *rescue clause*) [19].

  Our approach differs from the previous one in two points. First, the rescue clause is moved out from the methods to the relationship. As part of its semantics, a relationship can handle a failure on any of the related objects by looking for an alternative in the other object. Second, this approach is more object-based rather than method-based where the scope of the handler is the object, better said, the relationships in which the object participates. For instance, a conventional approach to face the previous exception on the *invoice* method, would have been to handle that exception by the caller method which in turn, can propagate the exception to its caller and so on. The propagation proceeds along the invocation chain. Now, that *invoice* exception can be handled by the initial receiver's related objects which can in turn propagate the exception to their related objects and so on. The propagation proceeds along relationship links till an object is reached which handles satisfactory the exception. Whereas what is an exception, is decided based on the method alone, how an exception is handled can be decided based on either the caller method (i.e. the traditional approach) or the object receiver (i.e. the relationship approach).

  Of course, for the approach to be safe, the alternative methods should obey the rule imposed by the system for method 'substitution' (i.e. the covariance or the contravariance rule). Hence, the system checks this rule when exception propagations are introduced at relationship creation time.

UPC
Facultat d'Informàtica
de Barcelona - Biblioteca

Furthermore, three additional aspects should be mentioned namely:

- the propagation of the operation can be conditioned to the satisfaction of a given predicate on either the relationship domain or the relationship range object. For example, in the *part_of* relationship, a common situation found in CAD/CAM systems is when the delete operation is propagated from the whole to a part *only if* the part does not participate in any other whole. Another example is the *working_in* relationship: expenses can de delegated provided the department budget is bigger than *10000*, and the employee status is *seller*.

- obtainment of the arguments of the induced operation. A reasonable alternative is to get these arguments from the triggering operation arguments.

- conflicts in the propagation graph. Such conflicts include possible cycles and ambiguity when several delegation alternatives are available. The later is similar to the name conflict which arise in multiple inheritance when a method can be inherited from different superclasses. The solution can be either a graph-oriented approach where an error is signaled if any conflict occurs, or a linear-like approach where conflicts are overcome by flattening the acyclic graph based on same ordering mechanism (e.g. the older is the relationship, the higher is its priority).

In what follows, it is shown how these ideas has been supported in ADAM [22], an object-oriented database system implemented in Prolog. First, the constructors used to describe the operational semantics are presented, and later, it is described how these constructors are realised.

3 Specification of update rules for relationships in ADAM

3.1 A brief overview of ADAM

In ADAM new objects are created by sending the message *new* to the class of which the object is to be an instance. For example, to create a new class called *person* which is an instance of the metaclass *entity_metaclass*, the following call is made:

```
new([person,[
  attribute(att_tuple(cname,global,single,total,string,[])),
  attribute(att_tuple(sex,global,single,optional,string,[])),
  attribute(att_tuple(born_in,global,single,optional,string,[])),
  method((salary_amount(global,[],[],integer,[Amount]) :-
  ..........
]]) => entity_metaclass.
```

The argument of *new* is a Prolog list, the first element of which is the name of the object, and the second element of which is a list of the attributes of the object. An attribute has a name and it is described by several facets: the visibility, the cardinality, the status, the type and the constraints attached to this attribute (empty list in the above example). Methods to retrieve *(get_)*, to delete *(delete_)* and to change *(update_, put_)* attribute values are automatically created by the system, so that attributes are *always* handled by these methods.

When *new* is used to create an instance rather than a class, the first element in the list passed to *new* is unified with the system-generated unique identifier of the object, e.g. 4@person[2]. For example, to create an instance of the class *person* in the variable OID, the following message is sent:

```
new([OID,[
      cname([odile]),
      sex([female]),
      born_in([usurbil])
]]) => person.
```

For further details about ADAM see[22, 11].

ADAM has been extended to represent relationships as *first-class* objects so that the relationship semantics is no longer hidden within the related classes but explicitly defined.

In the former version [9], the relationship semantics included the domain, range, cardinality and own attributes of the relationship as well as system support for the inverse link constraint and additional constraints which could be provided by the user. For example to define the *setting_up* relationship class between a class *company* and a class *region*, the command shown in figure 1 is invoked.

Being classes, relationship definitions can involved the specification of attributes and methods as other entity classes. Additionally, the *related_class* statements describe the objects between which the relationship is established. Both statements have a *related_class_tuple* tuple as their value where the following items are specified:

- the name of the class participating in the relationship (e.g. *company* or *region*),
- the role played by the class in the relationship (e.g. *the_company* and *the_region*),
- the attribute-based view of the relationship. As in [25] relationships can be treated either as objects or as attributes on any of the participant classes. For instance, the company is 'seen' from the point of view of the region as the attribute *industries*,

[2]The identifier 4@person is an internal identifier. In practice one would use a variable *Baby*, which had been instantiated by another goal, e.g. *get_by_cname([odile],Baby) => person*, instantiates Baby with the object identifier of the *person* whose name is *odile*.

```
new([setting_up, [
   attribute(att_tuple(established_in,global,single,optional,integer,[])),
   method((___
   related_class1([
      related_class_tuple(company,the_company,sets_up_in,single,[])]),
   related_class2([,
      related_class_tuple(region,the_region,industries,set,[])]),

   operational_semantics([
      delegating get_region_name(_) in the_company
         with get_region_name(_) in the_region,

      reacting put_regulations([Regulation]) in the_region
         with is_it_enforced([Regulation]) in the_company
         if_to (Regulation = Type-_,
            get_activity(Type) => the_company),

      anticipating new_development([Dev]) in the_company
         with check_legislation([Dev]) in the_region
   ])
]]) => relationship_class.
```

Figure 1: The *setting_up* relationship.

- the cardinality of the relationship, either *single* or *multi*,
- a set of constraints on the object which participates in the relationship.

This definition has now been extended with the *operational_semantics* attribute whose values are propagations which specifies the update rules for the relationship.

3.2 Specification of update rules for user-defined relationships

Updates rules are specified according to the following syntax:

```
<update_rule> ::= <coupling_mode>  <triggering_operation> IN <role>
                        WITH <induced_operation> IN <role>
                        [IF_TO <condition>]
                        [IF_FROM <condition>]
```

```
new([administrative_part_of,[
   is_a([part_of]),
   related_class1([
      related_class_tuple(region,the_whole,formed_by,set,[])]),
   related_class2([
      related_class_tuple(region,the_part,included_in,single,[])]),

   operational_semantics([
      reacting further_investment([Inv]) in the_part
         with further_investment([Inv]) in the_whole,

      reacting put_regulations([Reg]) in the_whole
         with put_regulations([Reg]) in the_part
   ])
]]) => relationship_class.
```

Figure 2: The *administrative_part_of* relationship.

```
<coupling_mode> ::=  DELEGATING | REACTING | ANTICIPATING | EXCEPTION_ON
```

The *IF_TO* and *IF_FROM* conditions allow to specify predicates on the object receiver and the operation parameters of the triggering and induced operation, respectively. Such conditions should be satisfied for the propagation to occur.

The arguments of the induced operation are obtained from those of the triggering operation through instantiation. Two key words, the name of the roles, are provided to refer to the current domain and range objects (e.g. *the_company* and *the_region* in the previous example).

For instance, assume *region* objects hold a set of regulations to be obeyed by companies which want to set up in this region (e.g. environmental laws), and a method *check_legislation* can be sent to a *company* to check the fulfilment of the regulations by the companies. On the other hand, as time goes by, *company* objects can have *new_developments* (e.g. new buildings) and regions can approve new regulations to be obeyed by the companies. This situation can be reflected by enlarging the previous relationship with the update rules shown in figure 1. The update rules:

- support the delegation of the *get_region_name* from *the_company* to *the_region*

```
new([small_administrative_part_of,[
   is_a([administrative_part_of]),
   related_class1([
      related_class_tuple(region,the_whole,formed_by,set,[])]),
   related_class2([
      related_class_tuple(small_region,the_part,included_in,single,[])]),

   operational_semantics([
      exception_on get_representatives(Rep) in the_part
         with get_representatives(Rep) in the_whole
         if_to (exception = 'no_available'),
   ])
]]) => relationship_class.
```

Figure 3: The *small_administrative_part_of* relationship.

- assure that once a new regulation is introduced for a region, this regulation is enforced by the companies in the region. This propagation is restricted to companies whose activity coincides with that of the regulation,
- assure that before a company undertakes a new development, the region's regulations are checked. If any regulation is violated, the development cannot take place.

Being classes, relationships can be specialized, to form relationship hierarchies. This situation can be very useful in CAD/CAM contexts where well-know relationships (e.g. *part_of*) can be specialized to account for special requirements. Besides inheriting the update rules of the superclass, a specialized relationship can define its own update rules. As an example, consider a special case of the *part_of* relationship which relates a region (i.e. the *part*) with a higher-level administrative unit (i.e. the *whole*) (e.g. Great Britain and Europe, Scotland and Great Britain). Each administrative unit has autonomy to have its own regulations besides inheriting the regulations which come from the higher administrative unit in which it is included. We can be interested in assuring that any regulation on the *whole* must be obeyed in the *part*, and than any investment (e.g. from the E.E.C) in the *part* should be considered as an investment in the *whole*. This situation can be modeled by the relationship shown in figure 2. The *administrative_part_of* relationship is defined as a subclass of the *part_of* relationship where *further_investment* and *put_regulations* methods are propagated.

Hence, entity specialization can be parallel by specialization of relationships in which the

entities participate. Consider *small_region* is introduced as a subclass of *region*. Now, the features to be specialized include not only attributes and methods but also the relationship semantics in which this new subclass can be involved. For instance, each administrative (or political) unit have a different set of people's representatives. The representatives are not propagated (i.e. the representative in the European parliament are different from those in the British parliament). However, small administrative units -whatever they are- do not have their own representatives but they obtain them from their upper administrative units. That is, if the *get_representatives* method fails, rising the 'no available' exception, such exception can be propagated from the *part* to the *whole* only if the part is an instance of *small_region* class. Such situation is shown in figure 3 with the *small_administrative_part_of* relationship. The related classes are *region* and its subclass, *small_region*, and the additional operational semantics contemplates the delegation of representatives from the *small_region* to the *region*. Notice that if the object which plays the role of *region* is in turn, a *small_region*, it will delegates the message to its upper region. And so on, till a 'big' region is reached and its representatives obtained. If the difference between *region* and *small_region* only stem from whether they have representatives or not, such difference can be account for within the relationship rather than being hidden and distributed among user programs.

Thus, the relationship semantics can be customized to fit the requirements of the designer instead of having a fixed system-provided semantics.

4 Supporting update rules through event-condition-action rules

Supporting relationships poses two main problems. First, its inherent non-local nature since it affects the two related objects. This has led to some implementations where the relationship semantics is split between the related objects. Besides impeding the maintenance, such approach is not akin with the ideas defended by the object-oriented paradigm whereby information on an entity should be grouped together with the entity.

A second difficulty stems from relationships being defined long after the related classes were created. With the 'splitting' approach, this would mean that the designer has to be aware of the relationships a class participates in.

What is required is a mechanism able to both monitors the related classes -without interfering with them- and react somehow based on this monitoring. Such mechanism is available in the so-called *active database systems* [15]. For object-oriented DBMSs, event-condition-action rules (henceforth referred as ECA rules) have been proposed in [8] to support this active behaviour, i.e. behaviour exhibited automatically by the system in response to events generated internally or externally, without user intervention.

```
new([EventOid,[
   active_class([person]),
   active_method([put_age]),
   when([before]),
]) => message_sending_event.

new([RuleOid,[
   event([EventOid]),
   condition([(
      occurrence_params([_Method,_Object,[PersonAge]]),
      PersonAge > 130
   )]),
   action([(
      occurrence_params([_Method,ThePerson,[PersonAge]]),
      get_cname(PersonName) => ThePerson,
      writeln(['The person ',PersonName,'with age ',PersonAge,..]),
      fail
   )]),
   is_it_enabled([yes]),
   disabled_for([1@person,23@person])
]) => integrity_rule.
```

Figure 4: A rule to prevent people from being older than 130.

ADAM is an active DBMS where rules and events are seen as first-class objects which are described using attributes and methods [10]. Rule structure is essentially defined by the **event** that triggers the rule, the **condition** to be checked and the **action** to be performed if the condition is satisfied. The condition is a set of queries to check that the state of the database and/or the event occurrence parameters are appropriate for action execution. The action is a set of operations and may have a variety of aims, e.g. enforcement of integrity constraints, provision for user intervention, propagation of methods, etc. Information about the operation occurrence which causes the rule to fire is available for use in both the condition and action definitions via the system-provided predicate *occurrence_params* which has the name of the method, the object receiver and the method arguments as its parameters.

Events are classified according to their generator (e.g. the message-sending mechanism, the interruption mechanism, the clock). Events risen by message-sending are described

by the following attributes: the **active_method** (i.e. the method firing the rule), the **active_class** (i.e. class of the instances which react to the active method) and the **when** (e.g. *after* or *before* the active method is executed, *failure* of the method, or *instead_of* the method) attributes. In figure 4, an example is given of an active rule that prevents people from being older that 130. This rule will fire *before* executing the *put_age* method. The condition checks whether the argument of the method (i.e. the new age of the person) is greater than 130 or not. If this condition is not met, the rule is not applicable and the action is ignored. Method execution can continue as normal. Otherwise, the rule's action *is* executed. In the case of our example rule, the action displays a message and then fails, preventing *put_age*, and therefore the update, from proceeding. It is also possible, if require, to pass a value from the condition to the action, by using the *condition_result* predicate, the argument of which is instantiated with any value required after condition evaluation. This can be useful in avoiding the need to redo work in the action, that has already been performed during condition evaluation.

In addition to the event-condition-action description, two more attributes are needed to specify the status of the rule itself, i.e. whether it is enabled or disabled. The attribute **is_it_enabled** describes the status at the level of the whole class appearing as the *active_class* of the rule, whereas the **disabled_for** attribute describes the status for specific instances of this class. In the above example, the rule is enabled for all instances of the class *person* (because the value of *is_it_enabled* attribute is *true*) except for those with object identifiers *1@person* and *23@person*. Thus, a rule will not be fired if either the *is_it_enabled* attribute is *false* or the object identifier of the current object appears as one of the values of the *disabled_for* attribute.

Event and rule objects are automatically generated by the system in order to support the update rules specified by the designer. The next two sections are devoted to how events and rules are obtained from an update rule such as:

```
CouplingMode TriggeringOp  IN  TriRole
     WITH    InducedOp     IN  IndRole
     IF_TO   TriggeringCond
     IF_FROM InducedCond
```

In figure 5 the resulting event and rule objects are shown where *update_rule* is a subclass of ECA rules.

4.1 The event object

The event object can be obtained as follows

- *active_class,* is the class playing the role of *TriRole* (named as *TriggeringClass*). Such class is obtained from the *related_class1* attribute of the relationship class,

```
new([EventOid,[
   active_class([TriggeringClass]),
   active_method([TriggeringOp]),
   when([When])
]]) => message_sending_event,

new([RuleOid,[
   event([EventOid]),
   condition([(
      occurrence_params([_,TriggeringInst,Arguments]),
      condition_result(InducedInstS),
      % if ... the TriggeringInst satisfies the condition
      TriggeringCond,
      ---
      % obtains the InducedInstS related with TriggeringInst
      % which satisfied InducedCond and
      % which are not yet in the propagation cycle
      findall(InducedInst,
         (PossibleInstances,
         instance_of(InducedInst,InducedClass),
         InducedCond,
         not get_already_propagated(InducedInst) => RuleOid),
      InducedInstS),
      not InducedInstS = []
   )]),
   action([(
      % then, propagated the action
      occurrence_params([_,TriggeringInst,Arguments]),
      condition_result(InducedInstS),
      (member(InducedInst,InducedInstS),
      put_already_propagated([InducedInst]) => RuleOid,
      PropagatedOp => InducedInst,
      fail ; true),
      ---
   )]),
   is_it_enabled([yes]),
   update_rule_for([RelationshipClass])
]]) => update_rule.
```

Figure 5: Supporting update rules through ECA rules.

- *active_method,* is the *TriggeringOp* method,

- *when* is obtained depending on the coupling mode *CouplingMode.* Reaction of a message occurs *after* the initial receiver has responded to it. Anticipation of a message happens *before* the initial receiver is allowed to respond to it. Delegation occurs when a different method is executed *instead_of* the method initially sent. And exception propagation can happen when an exception is arisen by the the *active_method.* In this case, the value *failure* is assigned to the attribute *when.*

4.2 The rule object

The rule attributes are obtained as follows:

- *event* holds the object identifier of the event which is obtained as described in the last section,

- *condition.* The condition for propagation is twofold. First the *TriggeringInst* object which is the receiver of the *TriggeringOp* method, must satisfied the *IF_FROM* requirement. Second, there must exist at least one object to which the message must be propagated which satisfied the *IF_TO* condition. Thus, the value of the *condition* attribute is a piece of Prolog code which verifies these two requirements.

- *action.* If the condition is satisfied then the rule action has the job of sending the message *InducedOp* out to all the appropriate objects. Rather than waste time rebuilding this list within the action (named as *InducedInstS*), we can reuse the version generated during condition evaluation by passing this list as the parameter of the system-defined predicate *condition_result.*

- *is_it_enabled* has *yes* as its initial value.

An additional attribute **update_rule_for** is included for update rules which keeps the corresponding relationship class. This attribute is *total* in order to support the existence dependence between the rule and the relationship class, i.e. when the relationship class deletion causes also to delete the associated update rules.

Unfortunately, there is a further complication to be considered. As mentioned earlier, a perfectly innocent-looking combination of specifications can result, in practice, in an infinite sequence of propagations. This loop can be illustrated by the *marriage* relationship with the *house_removal* method. When this method is sent to a member of the couple, *house_removal* is propagated to his/her partner which in turn, propagates it to his/her partner -which is the initial receiver of the message- and so on. Something must be done to prevent such undesirable effects.

In our approach this is achieved by adding a new attribute - the **already_propagated** attribute - which records the identifiers of those objects which have received the message at the current stage in the cycle of propagations. In building the list of candidate objects for propagation we must now take care to exclude objects *which have already received the message*, i.e. those whose identifiers are stored in the *already_propagated* attribute.

Due to the importance of maintaining a complete list of objects which have received the message being propagated, we are forced to have only **one** *comprehensive* version of the **already_propagated** attribute for each event. Although, it jeopardizes rule modularity, the easiest way to achieve this is to ensure that, for any given event, only one propagating rule exists. Therefore, whenever a new propagating rule is created, the system checks whether a rule already exists with the same event. If one does, the conditions of the two rules are merged to form a single, dual-purpose piece of code.

There are two points to notice about the maintenance of the *already_propagated* attribute:

- Objects are added to the attribute within the rule *action*, immediately before the message is propagated to them. Unfortunately, this does not insert the first object, the one to which the message was originally sent, and so this must be added separately, within the rule *condition*.

- At the end of each propagation cycle the *already_propagated* attribute must be cleared, in readiness for the next, which may involve a different set of objects. The end of the cycle is recognized by the fact that the current object is the object to which the message was originally sent, i.e. the first object contained in the *already_propagated* list.

For simplicity sake, these instructions are not shown in figure 1 but substituted by dots.

5 Related work

Designers soon recognized the core role played by relationships in the UoD. Indeed, the Entity/Relationship model [7], which obviously includes relationships as main primitives, is nowadays widely used. Further developments in this area lead to a broad spectrum of semantic data models where the stress is on providing a richer and well-defined set of abstract relationships. Such development was fueled by the need to tackle new areas than unlike traditional applications, require to capture more complex domains. VLSI design is a case in point whose specific relationships and their associated semantics were identified in [3]

However, very rarely these models were supported by a DBMSs. Relational systems lack a proper constructor for relationships which are supported either as attributes or relations.

This situation is improved in OO systems which aim to reflect more accurately the UoD, reducing the so-called semantic gap by providing similar set of primitives for both design and implementation. However, this advantage is severely compromised as relationships are not directly expressible (except for the *is_a* relationship) but buried into entities.

Several authors have proposed to extend the OO model with relationships as explicit primitives. Relationship classes agglutinate the attributes and methods of the relationship itself [25, 14, 5, 20, 9] as well as constraints on the related entities. Since constraints commonly involved different related entities, a natural place to hold these constraints are in the relationships which link these entities. Hence, the relationship can be seen as monitoring the related entities in order to keep the validity of the constraints [4, 1]. As a result, object encapsulation is not violated since constraint maintenance is kept in a single place, the relationship, rather than being distributed between the related entities.

In [14] the distinction between *vertical* (i.e. abstract relationships) and *horizontal relationships* (i.e. user-defined relationships) is also supported by an object-oriented knowledge base. Every attribute within the system is seen as an object. Relationships then, are a special kind of attribute that can have a richer associated semantics. such as the *the mathematical properties* of the relationship (reflexive, symmetrical and anti-symmetrical) and the nature of the link. The latter includes *the diffusion* property whereby some attribute values can be delegated to other objects through the relationship. In our approach the focus is on the behaviour rather than the object state.

In [13] the so-called *active semantic relationships* are proposed which *'express and enforce the constraints imposed on an object by the participation of the object in some group of related objects'*. Constraint enforcement is achieved through *'rules which cause a relationship to propagate the effects of a message received by an object to other related objects'* -hence the fact that relationships are qualify as *active*. From an implementation point of view, Doherthy's approach is very similar to ours since a rule-based mechanism is used but conceptually and unlike our approach, relationship are seen as restricting the behaviour of related objects. In Doherthy's own words, *'the interaction of objects is similar to the group behaviour of the members of a society. Although a member af a society mah be capable of a wide range of behaviors as an individual, the allowed social behaviours are restricted by social relationships'*.

Our approach is somehow orthogonal to Doherthy's one as relationship semantics is not seen as keeping inter-object constraints, but supporting its own operational semantics. Integrity constraints commonly refer to invariants to be maintained on the *object's state* whereas in this paper the stress is on the *object's behaviour* by the participation of the object in a relationship. We would like to think of this approach as a more optimistic view of what a society is by which individual behaviour can be enhanced and complemented by other society members' behaviour.

The work presented in this paper is also related with providing more powerful software sharing mechanisms. From this point of view, delegation as a sharing mechansim has been proposed for *actor-based* systems [16]. An *actor* can be defined as an isolated agent which knows how to perform a given task. Actors are independent and active objects that communicate freely with their neighbours to accomplish tasks. Sharing is achieved through *delegation*, i.e. when an actor receives a message it does not know how to answer, it forwards the message to another actor.

Delagation semantics is commonly embedded within methods of different actors and hence, it is obscured what semantics is being modelled. In [2] the delegation mechanism is governed by the evaluation of a set of production rules possessed by each object. When an actor receives a messsage, it evaluates its rule set to decide how this message has to be handled. Two advantages can be drawn from this approach. First, the implementation of delegation is declarative and centralized in a single location, instead of embedded in the methods. Second, delegation can be customized per object since each object possesses its own rule set. Unlike our approach, sharing is object-based and entity-based rather than class-based and relationship-based. However, we coincides on the motivation for having sharing supported as an independent mechansim from methods. Sharing of behaviour is a different concept from behaviour itself. Methods should not be used to support both concepts.

In [27] *the Treaty of Orlando* is presented where the different sharing mechanism are analysed and characterised along the following dimensions:

- *Static versus Dynamic:* is sharing determined when an object is created (static) or when the object receives the message (dynamic)?
- *Explicit versus Implicit:* are there primitives that allow the sharing to be explicitly fixed (explicit) or do the system primitives enforce a uniform mechanism (implicit)?
- *Per group or Per object:* is sharing specified for a set of objects (classes) or for individual objects (actors).

Inheritance can be seen as a kind of static, implicit and per group sharing mechanism, whereas delegation is dynamic, explicit and per object. Our approach could be qualified using this terminology as a kind of static, implicit and per group sharing mechanism, where here the group is based on participation in a given relationship.

6 Conclusions

To cope with the increase in complexity of the domains tackled by databases, more powerful semantics tools are needed. Relationships play a main role in the capture of this semantics

as it has been shown by early work on semantic data models, but have, however, been relegated to be simple pointers in most object-oriented systems.

It has been extensively advocated for relationships to be *first-class* objects where all the semantics of the relationship is collected instead of this semantics being distributed among different operations attached to distinct classes. In a behavioural OO system, this semantics includes the update rules.

Here a proposal has been presented to specify these update rules through a set of primitives. It is has also been shown how these primitives are realised in an active object-oriented database system through event-condition-action rules. Apart from the interpreted nature of ADAM, the operational semantics primitives presented do not require interpretation (or run-time resolution of names) except for the moment when the relationship is defined, but this moment is equivalent in a compiled system to the compilation process after the relationship is declared.

So far only binary relationships have been considered. Further research is also required to provide more sophisticated strategies to solve conflicts in the propagation graph. In the current implementation, conflicts are solved based on a naive strategy by which the older is the relationship, the higher is its priority. In any case, it is our experience that by declaratively specifying the operational semantics, relationship maintenance, legibility and easy of use are greatly improved.

From the reuse point of view, the proposed primitives present a framework which can receive encapsulated units that can be reused only by extension or by change of *public* features. The risks introduced by some implementations of inheritance as a mechanism for reuse (e.g. unrestricted access and modification of parent's features, unability to protect against children's erroneous changes -the *fragile base-class problem* [24]-) do not appear here where the behaviour is reused though a relationship

Acknowledgement
The author would like to thank Suzanne Embury for her encouragement throughout the development and implementation of these ideas. Thanks have also to be given to Ray Fernandez who provides valuable comments on an early draft of this paper.

References

[1] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases*, pages 565–575. Morgan Kaufmann, 1991.

[2] J. Almarode. Rule-based delegation for prototypes. In *Proc. Intl. Conf. on OOPSLA*, pages 363–370, 1989.

[3] D. Batory and W. Kim. Modeling concepts for vlsi cad objects. *ACM SIGMOD*, 10(3):322–346, 1985.

[4] M. Bouzeghoub and E. Metais. Semantic modeling of object oriented databases. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases*, pages 3–14. Morgan Kaufmann, 1991.

[5] S.E. Bratsberg. Food: Supporting explicit relations in a fully object-oriented database. In W. Kent R.A. Meersman and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction*, pages 123–140. North-Holland, 1991.

[6] I. Amy Chen and D. McLeod. Derived data update in semantic databases. In *15th Intl. Conf on Very Large Data Bases*, pages 225–2235. Morgan Kaufmann, 1989.

[7] P. Chen. The entity-relationship model - towards a unified view of data. *ACM TODS*, 1(1), 1976.

[8] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules are objects too: A knowledge model for an active object oriented database system. In K.R. Dittrish, editor, *Proc. 2nd Intl. Workshop on OODBS*, pages 129–143. Spring-Verlag, 1988.

[9] O. Diaz and P.M.D. Gray. Semantic-rich user-defined relationship as a main constructor in object oriented databases. In W. Kent R.A. Meersman and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction*, pages 207–224. North-Holland, 1991.

[10] O. Diaz, P.M.D. Gray, and N. Paton. Rule management in object oriented databases: a uniform approach. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 317–326. Morgan Kaufmann, 1991.

[11] O. Diaz and N. Paton. Extending ODBMS using Metaclasses. *IEEE Software*, 11(3):40–47, 1994.

[12] K.R. Dittrish. Object-oriented databases: the notion and the issues. In *Proc. 1st Intl. Workshop on Object-Oriented Databases*, 1986.

[13] M. Doherthy, J. Peckham, and V.F. Wolfe. Implementing relationships and constraints in an object-oriented datbase using a monitor construct. In M. Williams N. Paton, editor, *Proc. of the 1st Int. Workshop On: Rules In Database Systems*, pages 347–363. LNCS Springer-Verlag, 1994.

[14] J. Escamilla and P. Jean. Relations verticales et horizontales dans un modele de representation de connaissances. In A. Flory and C. Rolland, editors, *Nouvelles Perspectives des Systems d'Information*, pages 153–185. EYROLLES, 1990.

[15] E. N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, 1993.

[16] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[17] S.N. Khoshifian and G.P. Copeland. Object identity. In *Proc. Intl. Conf. on OOPSLA*, pages 406–416, 1986.

[18] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[19] B. Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object Oriented Software Engineering*, pages 1–50. Prentice-Hall, 1992.

[20] R. Nassif, , Qui, and J. Zhu. Extending the object-oriented paradigm to support relationships and constraints. In W. Kent R.A. Meersman and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction*, pages 305–330. North-Holland, 1991.

[21] E. Oxborrow, M. Davy, Z. Kemp, P. Linington, and R. Thearle. Object-oriented data management in specialized environments. *Information and Software Technology*, 33(1):22–30, 1991.

[22] N. Paton. Adam: An object-oriented database system implemented in prolog. In M.H. Williams, editor, *Proc. British National Conference on Databases*, pages 147–161. Cambridge University Press, 1989.

[23] J. Peckham and F. Maryanski. Semantic data models. *Computing Surveys*, 20(3):153–189, 1988.

[24] D. Pountain and C. Szyperski. Extensible software systems. *BYTE*, May:57–62, 1994.

[25] J. Rumbaugh. Relations as semantic constructors in an object-oriented language. In *Proc. Intl. Conf. on OOPSLA*, pages 466–481, 1987.

[26] J. Rumbaugh. Controlling propagation of operations using attributes on relations. In *Proc. Intl. Conf. on OOPSLA*, pages 285–296, 1988.

[27] L.A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The treaty of orlando. In F.H. Lochovosky W. Kim, editor, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press, 1989.