A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language

Antonio Albano
Dip. di Informatica
Università di Pisa
Corso Italia 40, 56100 Pisa, Italy
albano@dipisa.di.unipi.it

Giorgio Ghelli
Dip. di Informatica
Università di Pisa
Corso Italia 40, 56100 Pisa, Italy
ghelli@dipisa.di.unipi.it

Renzo Orsini
Dip. di Informatica ed Applicazioni
Università di Salerno
Baronissi (SA), Italy
orsini@udsab.unisa.it

Abstract

Object-oriented data models are receiving wide attention since they provide expressive abstraction mechanisms to model naturally and directly both structural and behavioral aspects of complex databases applications. In an object-oriented data model, a database is modeled in terms of objects grouped in classes, organized into subclasses hierarchies. Moreover, associations between entities are modeled by defining properties of objects whose value is the related object. However this way of modeling associations has several limitations which make the description of some aspects of associations unnatural. To overcome these limitations an object-relationship data model is proposed which supports both the mechanisms of an object-oriented data model and a separate mechanism to model explicitly associations and to express declaratively common constraints on them. Constructs to support this model for a statically and strongly typed object-oriented database programming language are defined.

1 Introduction

Object-oriented databases are becoming increasingly popular as a means to overcome the limitations of commercial DBMS. These limitations concern both the data models, which favour the efficient use of secondary memory at the expense of expressiveness, and the programming languages, which are unsuitable for the growing complexity of the applications due to the scarce integration of the data model abstraction mechanisms with those of the programming language.

The object-oriented programming paradigm is a very promising means to develop a new generation of DBMSs since both the problem of the expressivity of the data model, and the problem of integrating procedural and data modeling aspects can be tackled. The main features of object-oriented databases are discussed in [Atkinson 89] [Dittrich 90] [Zdonik 90]. Let us briefly recall some basic features of an object-oriented data model, and in particular how associations between entities are modeled. An object-oriented data model is based on the notions of *objects* and *classes*. Objects are used to model real world entities, and they have an immutable identity: the state of an object can be modified only by an object's own methods. Classes are sets of objects, used to model sets of homogeneous entities, and they are organized into a subclass hierarchy.[1] Associations between entities are modeled as properties of objects, i.e. as attributes whose values are the associated objects. A class is associated with an object type, in such a way that all the values of that object type, and only that particular object type, belong to the class. When objects in two classes a and b, whose elements have type A and B, are mutually related by an association, an attribute of type B is defined in the object type A, and vice versa, to model two one-directional relations. The constraint that the relation from a to b is the inverse of that from b to a, cannot be expressed declaratively in the object definitions, but must be coded in the methods which implement the association. The main advantage of this unification of attributes and associations is simplicity, since the same mechanism is used to deal with both of them, unifying the mechanism used to retrieve an attribute with the one used to retrieve associated objects, and the mechanisms to declare cardinality, surjectivity or non-modifiability constraints for both attributes and associations. However this approach has some limitations [Rumbaugh 87]:

- associations are conceptually a higher level abstract notion, their implementation should be decided by the DBMS; attributes, on the other hand, force the programmer to choose a specific implementation for them;
- the association semantics is split between different objects;
- associations are symmetric and the enforcement of the inverse relation constraint is not efficient when a method is used;
- associations are not necessarily binary, and they can have their own attributes; these aspects can only be modeled indirectly by means of attributes;
- associations relate objects which exist independently, and it ought to be possible to define them incrementally without having to redefine the structure of existing objects;
- operations on relationships as a whole are generally not straightforward.

This work was carried on with the partial support of Ministero dell'Università e della Ricerca Scientifica e Tecnologica, E.E.C. Esprit Basic Research Action 3070 FIDE, and of Italian C.N.R., P.F. Sistemi Informatici e Calcolo Parallelo.

[1]This use of the term class is different from the standard one in the context of object oriented languages, where it refers to the type of objects

Another fundamental problem with an object-oriented data model is the enforcement of the constraint that the extension of a class (the set of its elements) coincides with the set of all the elements of the related type. This "extension coincidence constraint" is used in the object-oriented data model to enforce the referential constraint as follows: when an attribute *p* of an object is used to model an association with objects which must be elements of a class *b*, this constraint is enforced by defining the type of the object attribute *p* to be that of the elements of class *b*. While enforcing the extension coincidence constraint when objects are created or inserted into a class is easy, enforcing it when objects are removed from classes is difficult. Removing an object from a class, by extension coincidence, is only allowed if the object can no longer be reached by the associated type. In any object oriented language this condition is undecidable, and the implementation of any reasonable approximation of it requires either a system-controlled implementation of associations or an extremely costly operation. A common alternative approach is to mark any object removed from a class as "killed", which allows raising a failure when that object is successively accessed; this is slightly better than leaving dangling references, but cannot be regarded as an enforcement of the referential constraint.

To overcome the above limitations in modelling association with an object-oriented data model, an object-relationship data model is presented. The main contribution of the proposal is the inclusion of the following features in a strongly typed, object-oriented database programming language:
- a construct to represent associations as *n*-ary symmetric relations among classes;
- associations can be organized into a specialization hierarchy;
- the referential integrity constraint is actually enforced;
- several constraints on associations, such as cardinality, surjectivity, dependency and non-mutability can be defined declaratively;
- the system is requested to implement associations, and the referential constraint can be enforced at a reasonable cost.

The focus will be on the mechanisms to model classes and associations. The other features of the database programming language, in particular the mechanism to model objects, is beyond the scope of this paper, and is given in [Albano 90].

The organization of the remainder of the paper is as follows. Section 2 briefly overviews the object-relationship data model. Section 3 discusses constructs to model classes of objects and associations between classes in an object-oriented programming language. Section 4 presents a minimal set of basic operations, where all the high level constructs can be interpreted. Section 5 compares related works. Conclusion comments on the work in progress.

2 The Object-Relationship Data Model

In the object-relationship data model, as in the object model, real world entities are modeled by objects collected in classes. Classes are sets of homogeneous objects, and inclusion and mutual disjointness constraints can be defined on classes. Associations between objects are not represented by the aggregation mechanism, but by *n*-ary relations relating objects in classes and also other values, which represent the attributes of the association. An inclusion relation can be defined on pairs of associations. The following constraints can be defined on associations:
- Surjectivity constraints can be defined for single components of associations. An association *AB* between classes *A*, and *B* is surjective (or total) on the class *A* if all the elements of *A* appear in *AB*.
- Dependency constraints can be defined between classes and associations. An association *AB* between classes *A* and *B* depends on the class *A* (*AB owned by A*), when, any time an object is removed from the class *A*, all the associations involving that object are removed. Class *A* depends on *AB* (*AB owns A*) if an object is automatically removed from *A* when the last tuple in *AB* involving that object is removed. An association can simultaneously own and be owned by a class. Classes and associations can own or be owned by any number of other associations and classes. *AB owned by A* and *AB owns B* together mean that, when the last element in *A* which refers to an element *b* in *B* is removed, then *b* is removed too. These constraints can be combined with keys and surjectivity to express the many different flavors of *composite objects* described in literature [Kim 89].
- Constancy constraints can be defined. In the *n*-ary case, the definition of constancy is: Let $A_1 ... A_n$ be some components of an association *Assoc*; *Assoc* is constant with respect to $A_1 ... A_n$ if, for any tuple $a_1 ... a_n$ of values for $A_1 ... A_n$, the set of tuples in *Assoc* with components $a_1 ... a_n$ is always the same in any state where $a_1 ... a_n$ exist (i.e. in any state where $a_1 ... a_n$ belong to their classes).
- Subsets of the components of an association can be declared as keys for the association. This means cardinality constraints can be expressed, e.g. distinguishing between single and multi valued relations.

The class and association constructs are exemplified through a slight modification of the simplified university administration application used in [Casanova 89], illustrated in Figure 1.

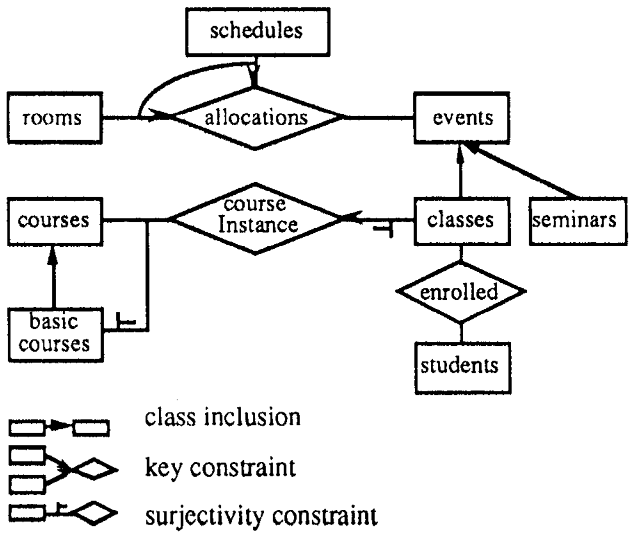

Figure 1: Examples of classes and associations

Schedules is a class of time intervals; *rooms* are the available rooms; *events* are the events, such as classes, seminars etc. that are scheduled; *courses* are the courses that can be offered; *basicCourses* are the fundamental courses which are always offered; *classes* are the classes running; *seminars* are the scheduled seminars; *students* are the registered students. The relationship *allocations* indicates which rooms are allocated for which events in which time intervals; *courseInstance* indicates the course of each class; *enrolled* indicates which students are enrolled in which classes. The following constraints are specified:

1) *inclusion constraints:*
- *classes* and *seminars* are disjoint subsets of *events* and *basicCourses* is a subset of *courses*;

2) *referential constraints:*
- insertions into *allocations*, *courseInstance* and *enrolled* fail if the objects used as components are not elements of the corresponding classes;
- the removal of an element *x* from the classes *rooms*, *events* or *schedules* fails if *x* appears in *allocations* (i.e. *x* is the value of a field of an element of *allocations*);
- the removal of an element *x* from the class *courses* fails if *x* appears in *courseInstance*;
- the removal of an element *x* from the class *classes* fails if *x* appears in *enrolled*;
- the removal of an element *x* from the class *classes* does not fail if *x* appears in *courseInstance*, but forces the removal of the element of *courseInstance* where *x* appears;
- the removal of an element *x* from the class *students* forces the removal of all the elements of *enrolled* where *x* appears;

3) *surjectivity constraints:*
- each element of *classes* is associated by *courseInstances* with at least one element of *courses*;
- each element of *basicCouses* is associated by *courseInstances* with at least one element of *classes*;

4) *key (cardinality) constraints:*
- each element of *classes* is associated by *courseInstances* with a maximum of one element of *courses*;
- each pair of elements of *rooms* and *schedules* is associated by *allocations* with a maximum of one element of *events*;

5) *constancy constraints:*
- each element of *classes* is always associated by *courseInstances* with the same element of *courses*;

For the sake of simplicity, in Figure 1 subclass disjointness, dependency and constancy constraints were not represented. This database is described by the following schema, using the constructs that will be defined later; the definition of the object types (like *ASchedule, ARoom*...) is out of the scope of his paper.

```
let schedules = new(classOf ASchedule)
let rooms = new(classOf ARoom)
let courses = new(classOf ACourse)
let basicCourses = new(classOf ABasicCourse
                                    are courses)
let students = new(classOf AStudent)
let events = new(classOf AnEvent)
let classes = new(classOf AClass      are events
                                      but not seminars)
let seminars = new(classOf ASeminar   are events
                                      but not classes)
let allocations = new(assocOf
                TheRoom: ARoom in rooms
                TheSchedule: ASchedule in schedules
                TheEvent: AnEvent in events
                ContactPerson: string
                key (TheRoom TheSchedule))
let courseInstance = new(assocOf
                TheCourse: ACourse in courses
                                  onto basicCourses
                TheClass: AClass owned_by classes
                                  onto classes
                constant_on (TheClass)
                key (TheClass))
let enrolled = new(assocOf
               TheStudent: AStudent owned_by students
               TheClass: AClass in classes )
```

In this model, the typical object-oriented mechanism of aggregation can still be used to build relations between objects, by defining object attributes with an object type.

This can be used, for instance, to model complex objects, i.e. objects with parts which can be viewed as independent objects, when these parts are not collected into another class. However, in this model if aggregation is used to define associations between classes, the referential constraint is not enforced.

3 Classes and Associations for the Object-Relationship Data Model

This section presents the constructs to describe classes, associations, and the constraints supported by the object-relationship data model in an object-oriented database programming language. The next section shows how the semantics of these constructs can be given in terms of a minimal nucleus of primitive operators.

In the language, classes are sets of data, and associations are relations, i.e. sets, of bindings, organized into a specialization hierarchy. The basic operations on classes and associations are creation, insertion and removal of elements. Constraints on classes and associations can be defined declaratively and are enforced by a general trigger mechanism which is described in the next section.

Classes are first class values and class types are first class types, like sequences and sequence types are. Consequently (*a*) classes can be used in any combination with the other data types constructors of the language to build complex structures, (*b*) no special naming mechanism is used for classes, and (*c*) the interaction of classes with other features of the language, such as modules functions or object types, is governed by the general rules of the language. The same is true for associations: classes are just the unary case of the general mechanism of *n*-ary associations. All the operators over classes and associations are statically and strongly typed. For any class the type of its elements is declared (the element type of the class), but the elements of a class are generally only a subset of all the values belonging to its element type.

Let us give first some definitions about subtyping, equality, bindings and signatures, since these notions are used to define associations, which are sets of bindings.

3.1 Subtyping, bindings, equality and signatures

In the complete language an inclusion hierarchy is defined on types, such that if A is a subtype of B, written $A \leq B$, any value belonging to type A also belongs to type B.

A binding is a set of pairs <label, value>, and a signature is a set of pairs <label, type>; in both cases all the labels are different. A binding *satisfies* a signature *Sign* if it contains all the labels of *Sign* and for any label in the signature, the value associated with that label in the binding belongs to the type associated in the signature.[2]

A binding is denoted as (**let** $l_1 = v_1$ ... **let** $l_n = v_n$). A signature ($l_1 : T_1$ ... $l_n : T_n$) is a supersignature of another one ($m_1 : U_1$ ... $m_m : U_m$) if the set of the labels of the first one is a subset of the set of the labels of the second, and if the type associated with any label in the super-signature is a supertype of the one associated with the same label in the subsignature.

In the complete language equality is defined on all the types. It is defined by identity (i.e. by creation time) on functions and updatable values (objects, updatable variables, classes and associations), and is defined by value on values belonging to the other non-updatable type constructors (bindings, variants and sequences).

Equality is type dependent on types where it is value defined: for example, if two bindings with labels a and b have the same integer value in field a and different integer values in field b, they are equal if compared with type (a:Int) and different if compared with type (a:Int and b:Int). On the other hand, equality is type independent in the other types: two objects are equal if and only if they have the same identity, whatever the type used to compare them.

Two types are *compatible* if there is a type V which is a supertype of both of them.

3.2 Classes and associations

A class is an ordered set of distinct elements with the same type, and an association is an ordered set of distinct bindings satisfying a fixed signature. Classes and associations are first class values of the language, and their structure is described by the first class types **Class**(*ElementType*) and **Assoc**(*Signature*) **key** $keylist_1$ ... **key** $keylist_n$. This means that associations can be used to form arbitrarily complex structures, although a scheme is usually defined by using associations with a flat structure, as shown in the next section.

Classes and associations are created empty, and then elements and bindings are inserted and removed. Classes and associations are inspected by using relational-like bulk data operators.

3.2.1 Creation

A new empty class is defined by the following operation:

2 Bindings are similar to records, although in the full language there is a distinction between these two type constructors.

new(**classOf** *ElType* **are** $C_1,..., C_n$
butnot $A_1,..., A_m$
beforeInsert *expr*
beforeRemove *expr*): **Class** *ElType*

new classOf creates a new class which collects elements of type *ElType* (usually an object type), and such that its extension is always included in that of the superclasses $C_1,..., C_n$ (*extensional inclusion constraint*), and it is always disjoint from that of the classes $A_1,..., A_m$ (*extensional disjointness constraint*). These constraints are maintained automatically, as shown in Section 4.

beforeInsert *expr* (**beforeRemove** *expr*) specifies that *expr* must be executed before the insertion (removal) of an element in (from) the class. The identifier *this* can be used inside *expr* to denote the element to be inserted (removed), and *self* to denote the *whole* class. This mechanism can be combined with the **assert** and **defer** constructs described in Section 4 to define pre and post conditions.

The declaration above is well typed only if the element types of the classes $C_1,..., C_n$ are all supertypes of *ElType* (*intensional inclusion constraint*). If B is the created class, $C_1,..., C_n$ are the *immediate superclasses* of B.

$A_1,..., A_m$ are classes which must never intersect B. If TA_i is the type of the elements of the class A_i, the clause **butnot** is well typed only if any TA_i is compatible with *ElType*. Both clauses **are** $C_1,..., C_n$ and **butnot** $A_1,..., A_n$ can be omitted. Examples are given in Fig.1.

A new empty association can be defined as follows:

new (**assocOf** *ExtSignature* **are** $A_1,..., A_j$
{**key** keylist}
{**constant_on** label [**in** class] ... label [**in** class]}
{**beforeInsert** expr}
{**beforeRemove** expr}):
Assoc *Signature* **key** $keylist_1$ ... **key** $keylist_n$

ExtSignature is a signature extended with a set of constraint specifications, described below, and *Signature* is *ExtSignature* without those specifications.

A *keylist* is a list of attributes of the signature, specifying the constraint that two distinct bindings in the association must differ in at least the value of one attribute for each keylist.

constant_on is a constancy constraint, explained later.

beforeInsert *expr* (**beforeRemove** *expr*) specifies that *expr* must be executed before the insertion (removal) of an element in (from) the association. The labels of the association signature can be used inside *expr* to denote the fields of the binding to be inserted, and *self* to denote the *whole* association.

The **new assocOf** operation builds a new empty association to collect bindings with the specified signature. The extension of the new association is a subset of that of $A_1,..., A_j$; *Signature* must be a subsignature of those of $A_1,..., A_j$, and the keylists of the new association must imply those of $A_1,..., A_j$. An implication relation is defined on sets of keylists by the transitive and reflexive closure of the following rules (where the single keylists and the list of the keylists are regarded as sets):

$$\textbf{key list}_1 \ldots \textbf{key list}_n \rightarrow \textbf{key list}_1 \ldots \textbf{key list}_{n-1}$$
$$\textbf{key}\ (\text{list}_1) \ldots \textbf{key list}_n \rightarrow \textbf{key}\ (\text{list}_1, a) \ldots \textbf{key list}_n$$

A description of the constraints which can be declared in *ExtSignature* follows:

ExtSignature ::= {*label* : *Type* {*constraint* }}
constraint ::= **in** *class* | **are** *class* | **owned_by** *class*
| **onto** *class* | **owns** *class*

3.2.2 Referential constraints

label: *Type* **in/are/owned_by** *class*

The attributes of a binding of an association are divided into *components*, which are attributes whose values must belong to a specified class, and *association attributes* (*attributes* in short), which have no such constraint.

There are three different declarative ways of defining a referential constraint: **in**, **are** and **owned_by**.

These three clauses specify the same referential constraint to be maintained with different styles, i.e. either by raising failures or by forcing its satisfaction.

In more detail, the **in** clause means that: *a*) when a new binding is inserted in the association, a failure is raised if the value of the component is not contained in the specified class; *b*) when an element is removed from the referred class, the operation fails if the element is a component of a binding in the association.

The **are** clause means that: *a*) when a binding is inserted in the association, the value of the component is inserted in the class; and *b*) when an element is removed from the class, all the bindings with that element as a component are removed from the association.

The **owned_by** clause means that: *a*) when a new binding is inserted in the association, a failure is raised if that component is not contained in the specified class, as happens with the **in** clause; *b*) when an element is removed from the referred class, the bindings referring that element are removed, as happens with the **are** clause. So **owned_by** codifies a dependency constraint, and a dependency of the association from the class.
For the referential constraints **in** and **owned_by**, as for the surjectivity and constancy constraint, the element type of the referred class must be compatible with that of the attribute in the signature, even though they are usually exactly the same type. For the referential constraint **are** the type of the component must be a subtype of the element type of the referred class.

3.2.3 Surjectivity constraints

label: *Type* **onto/owns** *class*

While the referential constraint specifies that the existence of a binding in an association implies the existence of a value in a class, the *surjectivity* (or *totality*) constraints enforce the converse implication: the existence of elements in some classes necessitates the existence of a binding involving them in some association.

The **onto** clause corresponds to the **in** clause; it means that when an element is inserted in the class, then in the same transaction a binding referring to that element must be inserted in the association, and conversely, when the last binding in an association referring to an element is removed, then in the same transaction that element must be removed too. The complete language supports nested transactions, as specified in the next section, and this constraint is checked at the end of the transaction where the class insertion or the association removal take place. Elements are supposed to be inserted first into classes and then into associations, and conversely for removal. Referential constraints can thus be checked immediately, whereas surjectivity constraints can only be checked at the end of the smallest enclosing transaction.

The **owns** clause is the surjectivity counterpart of the referential **owned_by** clause: like **onto**, when an element is inserted in the class, then in the same transaction a binding referring to that element must be inserted in the association, but when the last binding referring to an element is removed from an association, that element is removed from the class too, at the end of the transaction.

In object-relationship schemes, any component has one referential constraint, but it can have zero, one or more surjectivity constraints, defined on different subclasses of the class of the referential constraint (see the *courseInstance* association in Figure 1).

In the following tables the precise relationships between the above constraints are summarized. Tables 2 and 3 highlight that in fact a fourth kind of referential constraint could have been defined, characterized by the behaviour "class.**insert**(x) - **fail**" (Table 2). But in the surjectivity family there are just two possible constraint, since there is no alternative to failure if there is class insertion (Table 3).

Table 1: Conditions Enforced and Operations Monitored by the Constraints

Constraint	Enforced condition	Monitored operations
referential	$x \in$ assoc.label $\Rightarrow x \in$ class	assoc.**insert**, class.**remove**
surjectivity	$x \in$ class $\Rightarrow x \in$ assoc.label	class.**insert**, assoc.**remove**

Table 2: Action requested by a referential constraint, before performing an insertion/removal operation

Constraint	**assoc.ins(label=x...) if $x \notin$ class**	**class.rem(x) if (label=x...)$\in$ assoc**
in	**fail**	**fail**
owned_by	**fail**	assoc.**rem**(label=x)
are	class.**insert**(x)	assoc.**rem**(label=x)

Table 3: Action requested at commit time by a surjectivity constraint, if an insertion/removal operation is performed

Constraint	class.ins(x) if (label=x...)$\notin$ assoc	assoc.rem(label=x) if (x)$\in$ class
onto	**fail**	**fail**
owns	**fail**	class.**rem**(x)

3.2.4 Constancy constraints

constant_on *label* [**in** *class*] ... *label* [**in** *class*]

When an association is constant on one component-class pair *label-class*, all the bindings involving a value *el* of *label* can only be inserted in the association when *el* is inserted in *class* (or rather in the same transaction). Any other attempt to insert or remove associations involving *el* would fail, apart from the final binding removal which can take place only when *el* is removed from its class.

Constancy on a list of pairs $label_i$-$class_i$ means that the bindings involving a binding

$\{<label_1, el_1>, \ldots, <label_n, el_n>\}$

must be inserted in the association at the same time as the last el_i is inserted in its *class*, and can only be removed at the same time as at least one of the el_i is removed from its class. Constancy on many lists is just the conjunction on all the associated conditions.

This constraint is orthogonal to the cardinality (*key*), referential and surjectivity constraints. Its type rule specifies that the type of any *label* in the signature must be compatible with the element type of the corresponding *class*. If exactly one referential constraint is specified for a component, the clause *in class* can be omitted in the constancy constraint, and the class specified in the referential constraint is assumed. The constraint is well typed if the type of the components is compatible with the associated classes.

The constancy constraint completes the list of the declarative constraints which can be specified on classes and associations. In the next subsection the operators to update classes and associations are presented.

3.2.5 Updating operators

class.**insert**(elem), class.**remove**(elem)
assoc.**insert**(binding), assoc.**remove**(binding)

The **insert** operation takes a value of the element type of the specified class, executes all the declared constraint checking and automatic insertions (if *class* has some superclasses) and finally inserts the element in the class. If the argument of **insert** is already contained in the class, the operation has no effect.[3] **insert** is atomic, which implies that if a failure is raised during its execution, all its effects are undone. If **insert** causes an automatic insertion in a superclass, the constraints and automatic insertion of the superclass are executed too. **insert** behaves exactly in the same way on associations.

The **remove** operation on associations takes a binding whose signature is compatible with the association, and removes all the binding in the association which match the argument binding. It verifies all the associated constraints for all these elements, executes all the automatic removals, and finally removes them from the association; like **insert**, **remove** is atomic. On classes, it takes a value whose type is compatible with the elements of the class, and removes all the elements in the class which match the argument. In both cases, **remove** just removes the argument from the class/association, without destroying it, so that if that object/binding is accessible in some other way, it remains accessible after the removal. This is not a problem since the enforcement of the referential constraint does not depend on the coincidence of the extension of a class with the set of all the elements of the associated type.

Two values *a* and *b* belonging to the compatible types *A* and *B* "match" if they are equal with respect to any of the minimal common supertypes of *A* and *B*. In practice, if *a* and *b* are objects, they match if they have the same identity, whereas if they are bindings they match if the values associated to the common labels match. For example, referring to the example in Section 2, the binding (**let** *TheEvent* = *x*) matches all the bindings in the association *allocations* whose field *TheEvent* is equal to *x*.

3.2.6 Associative access operators

assoc.**has**(binding), assoc.**get**(binding)
class.**has**(value), class.**get**(value)

has, like **remove**, receives a binding whose signature is compatible with that of the association, and returns **true** if the association contains a binding which matches its argument. On classes, it receives a value whose type is compatible with the element type of the class, and returns **true** if the class contains an element which matches its argument.

The type constraint for the **get** operator is slightly different. On classes, it receives a value of a type which is not only compatible with the element type of the class, but also has the same equality, which means that for any pair of elements belonging to both types, they are equal when compared in one type if they are equal when compared in the other one. Any two compatible object types have the same equality, since objects are compared by identity. **get** returns the unique value in the class which is equal to its argument, and fails if no such value exists. **get** can be used to perform a sort of run-time type coercion: let *Student* be a subtype of *Person* and (*students:* **Class** *Student*) be a subclass of (*persons:* **Class** *Person*), and suppose that *john* has type *Person*. If *students*.**get**(*john*) does not fail, then it returns the same object as *john*, but with type *Student*.

On associations, **get** receives a binding whose signature *GetSign* satisfies the following constraints: (*a*) *GetSign* is compatible with the signature *AssocSign* of the association; (*b*) the set of labels of *GetSign* includes a key of the association; (*c*) for all the labels belonging to both *GetSign* and *AssocSign*, the associated types have the same equality. **get** returns the unique binding in *assoc* which matches the specified binding, and fails if no such binding exists. The conditions (*b*) and (*c*) imply that a maximum of one element of the association matches the **get** argument.

3.2.7 Relational-like algebra

In the full language, a sequence type Seq exists, with a set of relational-like operators, transforming sequences into sequences.

[3] This is enough to maintain the *set constraint*, i.e. the constraint that all the elements of a class are different; in fact, since all the updatable entities of the language are compared by identity, the set constraint cannot be violated as a side effect of an update operation.

A type Assoc*(Signature keylists)* is a subtype of the type **Seq** *Signature*, and a type **Class** *ElType* is a subtype of the type **Seq** *ElType*, so that the relational-like operators of the language can be applied also to associations and classes.[4] The abstract syntax of the relational-like operators is listed below. They are divided into the group of the operators defined on all sequences, which can be applied to both classes and associations, and those defined only on sequences of bindings, which can only be applied to associations.

General operators on sequences:

R **union** S, R **intersect** S, R **diff** S
R **select** condition
R **map** function
R **sort** sortList

The meaning of the operators is specified by their name; their type constraints are specified below, supposing, where needed, that *S* and *R* have type **Assoc** (*Signature keylist*...) or **Class** *ElType*.

union, **intersect** and **diff** can be applied to any pair of lists with a common supertype, returning a result in that type.

In **select**, *condition* is a function of type *Signature→Bool* (*ElType→Bool*)

In **map**, *function* is a function of type *Signature→T* (*ElType→T*), applied to all the elements of the association (or class) *R* to obtain a sequence of type Seq *T*.

In **sort**, *sortlist* is a sort condition for the element type of the sequence *R*; see [Ghelli 90a] for the precise language used to express sort conditions.

Operators on sequences of bindings:

R **project** labelList
R **times** S
R **rename** renameList
R **groupby** groupList

In **project**, *labelList* is a subset of the list of the labels of *R*.

times takes two sequences of bindings with disjoint sets of labels, and returns a sequence of bindings containing the union of the labels of the arguments.

In **rename**, *renameList* is a binding such as the expression **let** $n_1=o_1 \ldots n_m=o_m$ where $o_1,\ldots, o_m$ are labels of the original relation and $n_1,\ldots, n_m$ are all mutually different labels not included in the labels which remain in the relation after the o_i labels are removed.

In **groupby**, if *Signature is equal to*

$$l_1{:}T_1 .. l_m{:}T_m\ m_1{:}U_1 \ldots m_m{:}U_m$$

and *groupList* is equal to $l_1..l_m$, then *R* is partitioned in sequences where all the fields $l_1..l_m$ have the same value, each of these subsequences is transformed into just one binding of signature

$$(l_1{:}T_1 .. l_m{:}T_m\ m_1{:}\mathbf{Seq}\ U_1 \ldots m_m{:}\mathbf{Seq}\ U_m)$$

and the resulting sequence of type

$$\mathbf{Seq}\,(l_1{:}T_1 .. l_m{:}T_m\ m_1{:}\mathbf{Seq}\ U_1 \ldots m_m{:}\mathbf{Seq}\ U_m)$$

is returned.

Two operators, **makeClass** and **makeAssoc***(keylists)*, are defined to transform sequences and sequences of bindings into classes and associations, though these operators are usually not needed.

4 The Kernel Language

In the previous section a language was presented which supports the structure and the constraints of the object-relationship data model. This section shows that the language can be fully interpreted in a minimal kernel, built around (*a*) the general failure handling mechanisms of the language, (*b*) a simplified association mechanism with no predefined constraint declaration, and (*c*) a general purpose constraint maintenance mechanism for associations. In this way a formal semantics is given for the constraints presented, and for any possible combination of them. Besides this, the general purpose mechanism defined here is present in the language together with the specialization presented in Section 3, to allow the programmer to specify different constraints, or different flavors of the same constraints. The most appealing feature of the basic mechanism presented here is its simplicity, built around just one type operator and seven value operators.

In Section 4.1 the general failure handling and nested transaction mechanism of the language are outlined; in Section 4.2 the basic association mechanism is defined. In Appendix A the semantics of the declarative constraints is presented by giving their translation into the basic mechanism.

4.1 The transaction and failure mechanism

The full language supports nested transactions and a nested failure management mechanism, based on the following operators.

Failure management operators:

let exc excname: type;
failwith excname=value
assert boolexpr **elsefail** excname=value
try expr **exc** $\text{excname}_1=\text{var}_1$ **do** handler_1
  ...
  exc $\text{excname}_n=\text{var}_n$ **do** handler_n
  [**else do** handler]

let exc introduces a new exception name *excname* which is associated with values of type *type*.[5]

failwith raises an exception with name *excname* and value *value*; *excname* has been previously introduced by **let exc**. The exception propagates along the dynamic activation chain until an exception handler, defined using the **try** construct below, is found.

assert is equivalent to **if** *boolexpr* **then** *nil* **else failwith** *excname=value*; if the clause **elsefail** is omitted it fails with *failure=nil*.

try executes *expr*; if it fails with a name $excname_i$ then **try** executes the handler $handler_i$ binding var_i to the value of the exception. If the exception name is different from all the names $excname_i$, then there are two possibilities: if the **else do** branch is defined then the

[4] Associations and classes are ordered by insertion time; this ordering means thay can be viewed as sequences.

[5] Type is a ground type built without using object types.

corresponding general handler is executed, otherwise the exception is propagated.

Nested transactions operators:

atomic expr, **defer** expr, **old** assoc

atomic executes *expr*, and if it fails, before propagating the failure, rebuilds the state as it was before executing *expr*. In more detail, it undoes all the variable updates, the class/association insertions and removals and the effect of the operation **extend**, defined in the full language, which changes the type of an object without affecting its identity. **atomic** is a nested transaction mechanism; the outermost **atomic** is the transaction used for concurrency control.

A list of expressions to be executed before committing is associated with any transaction; **defer** is used to build this list, adding the specified expression to the end of the list of the current transaction. **defer** is used typically to defer the control of an integrity constraint to the end of a transaction.

old applied to a variable or to an association (or class) returns *a copy* of the value of that variable or association at the beginning of the current transaction. It is used to check dynamic integrity constraints.

4.2 The basic association mechanism

In the kernel of the language, classes are not defined, but only associations are. An association is only characterized by a signature and a list of keylists, without inclusion hierarchies or constraints.

The primitive creation operation for associations is:

new (**assocOf** Signature **key** keylist$_1$ ... **key** keylist$_n$):
    Assoc Signature **key** keylist$_1$ ... **key** keylist$_n$

On the basic associations a general purpose constraint verification/enforcement mechanism is defined. Any association contains two lists of function, called the *insertion* and *removal pre-operations*, which are applied to each binding which is inserted in an association, or is removed from it. If the argument of the **insert** operation is already contained in the association, the pre-operation is not executed. These pre-operations are defined with the following operators:

assoc.**beforeInsert**(**fun**(Signature) expr[6])
assoc.**beforeRemove**(**fun**(Signature) expr)

beforeInsert/beforeRemove add the function **fun***(Signature) expr* at the head of the insertion/removal pre-operations lists; *Signature* is the signature of the association.

insert, **remove**, **get** and **has** are defined on the basic associations as they are on the sugared version. These operators complete the definition of the basic association mechanism.

4.3 Translating classes and predefined constraints into the basic calculus

The complete translation of constraints is given in Appendix A — here, only the translation of classes is specified.

[6] Actually any function of type Signature→Type is accepted as an argument by *beforeInsert* and *beforeRemove*.

A class of type **Class** *Type* is translated as an association of type **Assoc** *(label: Type)*, where the label is arbitrary, and never used. The operations on classes are a syntactic abbreviation of the operations on associations, where a type *ElType* substitutes a signature *label:ElType* and values of type *ElType* substitute bindings satisfying *label:ElType*:

new (classOf Type): **Class** Type →
    new (assocOf label: Type): **Assoc** Type

class.**insert/remove**(value) →
    assoc.**insert/remove**(**let** label = value)

class.**beforeInsert/Remove**(func) →
    assoc.**beforeInsert/Remove**(
        fun(bind:{label:Type}) func(bind.label))

class.**get/has**(value) →
    assoc.**get/has**(**let** label = value)

5 Related Work

The relation mechanism in data models has been extensively studied since the proposal of the relational and entity-relationship data models. A recent proposal in the field of entity-relationship model, similar to the one presented here, is in [Casanova 89], where a data definition language is presented for an extended E-R model. The language allows entity sets to be defined as well as relationship sets. Relationship sets can have keys and surjectivity constraints. Besides general assertions, mutual exclusion and referential integrity constraints can be specified both on entity and relationship sets. These assertions are complemented by the facility to specify the existence of triggers, immediate or deferred, on operations. Only the conditions for firing triggers are described, not the triggers themselves, since the paper does not propose a particular data manipulation language. Entity and relationship sets can be organized into a specialization graph, to provide both inheritance and inclusion hierarchy.

While this work is similar to ours from a data modeling point of view, our proposal is expressed in the framework of a full language, which is both object-oriented and strongly and statically typed. Moreover, our kernel language is conceptually simpler and more regular.

From the field of object-oriented languages, both [Rumbaugh 87] and [Diaz 90] present a proposal to enhance object-oriented languages with a construct to represent user-defined relations. Rumbaugh was the first to stress the relevance of a relation construct in this context. His proposal allows *n*-ary relations to be defined over objects, but only with simple cardinality constraints. The language presented is untyped, and no specialization is defined over associations. Implementation issues are discussed together with the description of an actual implementation in a production-quality programming system developed by the author at General Electric. The proposal in [Diaz 90], expressed in the framework of knowledge representation language based on frames, presents a construct to define binary relationships between objects, with assertions and attributes which belong to the relationship as well as assertions and attributes added to the objects for as long as they participate in the relationship. In addition, surjectivity, cardinality and dependency constraints can be specified on relationships. Relationships are objects which can be specialized, and whose methods for creation, retrieval and updating can be modified. General constraints are intended as invariants to be preserved in the database: a complex system executes this task. In this language there is no concept of type or type checking, and, like in the Rumbaugh proposal, there are no retrieval or other bulk operators on relations. The constructs proposed are embedded in a high level object oriented extension of Prolog.

In [Atkinson 91] a new type constructor called *map* is presented. Whereas associations are inspired by the mathematical notion of finite relation, maps are inspired by the notion of finite function. The expressive power of the two notions is similar, since associations can be seen as maps without a range, while maps can be seen as associations with just a key. An interesting characteristic of this proposal is that both an ordering and an equality specification for the key can be declared together with the map type, and then become part of the type. The only constraint which can be defined on a map type is a form of constancy constraint. Many operations are provided to access and modify elements of maps, either singularly or by iterating over a specific subset of a map. An algebra over maps is defined, through classical operations on sets as well as through an operator similar to comprehension. Associations are proposed mainly as a data modelling abstraction mechanism, and for this reason they have a rich set of constraint specifications and can be organized into a specialization hierarchy. Maps, on the other hand, are also proposed both as an efficient bulk structure for database programming languages and as the data format of a canonical store manager for complex structured data. Maps could thus be used as a low-level structure to implement classes and associations efficiently, as well as associative data structures on them.

6 Conclusions

A mechanism has been defined to represent classes and associations in a database object oriented language. This proposal stems from the experience gained in designing, implementing, and using the Galileo database programming language [Albano 85]. It is characterized by the following features:

- Associations are not described by aggregation, as in the standard object oriented data models, but by a separate mechanism. With this approach the implementation choices about associations are left to the DBMS.
- Classes and associations are first class values of the language, and their structure is described by a first class type. This means that these constructors can be combined in any way with the other data type constructors of the language.
- The following constraints can be defined declaratively: class and association inclusion, key, referential, surjectivity, dependency and constancy constraints.
- All the above constraints are formally defined in terms of a minimal kernel based on just one data type constructor (**Assoc**).
- All the constructs presented permit a strong type checking (no type error is raised at run time by a well typed expression) which can be performed completely at compile time.

The mechanisms presented are included in a complete database programming language, which is currently under implementation, with the following features [Albano 90]:

- it is statically and strongly typed;
- it supports a module mechanism for structuring complex schemes and applications;
- it supports all the features of an object oriented language: object identity, state and methods encapsulation, type inclusion, multiple inheritance;
- it supports an object mechanism with separation between interface and implementation of an object type definition, and with an operator to change the type of an object dynamically without affecting its identity.

6.2 References

Albano A., L. Cardelli and R. Orsini, "Galileo: a Strongly Types Interactive Conceptual Language", *ACM Trans on DataBase Systems*. 10 (2), pp. 230-260, 1985.

Albano A., Ghelli G. and Orsini R., "Objects and Classes for a Database Programming Language", Tech. Rep. 5/24 Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, CNR, Roma, November 1990.

Atkinson M.P., Bancilhon F., DeWitt D., Dittrich K., Maier D., and Zdonik S., "The Object-Oriented Database Manifesto", *Proc. DOOD 89*, Kyoto, Japan, 1989.

Atkinson M.P., Lécluse C., and Richard P., "Bulk Types for Data Base Programming Languages: A Proposal", submitted for publication, 1991.

Casanova M.A., Tucherman L., Gualandi P.M., Pacheco A., and Cavalcanti M.R., "A Data Definition Language for Extended Entity-Relationship Model", Rio Scientific Center, Technical Report CCR-072, 1989.

Diaz O., and Gray P.M.D., "Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Database", *Conf. on Object-Oriented Databases*, Windermere, UK, 2-6 July 1990.

Dittrich K., "Object-Oriented Database Systems: The Next Miles of the Marathon", *Information Systems*, Vol. 15, N. 1, pp. 161-167, 1990a.

Ghelli G., "*Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*", PhD Thesis, TD-6/90, Dipartimento di Informatica, Università di Pisa, Italy, 1990b.

Ghelli G., and R. Orsini, "Types and subtypes as partial equivalence relations", In *Inheritance hierarchies in Knowledge Representation*, Lenzerini M., Nardi D, Simi M. (eds.), J. Wiley & Sons, Chichester, England, pp.191-209, 1991.

Kim W., Bertino E., and Garza J., "Composite objects revised", *Proc. ACM SIGMOD Conf. Management Data*, Portland, OR, June 1989.

Rumbaugh J., "Relations as Semantic Constructs in an Object-Oriented Language", *OOPSLA'87*,466-481, 1987.

Zdonik S.B., and Maier D., "Fundamentals of Object-Oriented Databases", in *Readings in Object-Oriented Database Systems*, Zdonik S.B., and Maier D. (eds), Morgan Kaufmann Publishers, San Mateo, CA, 1990.

Appendix A: The translation of the constraints

In this appendix the constraints presented in Section 3 are translated into the basic language presented in Section 4. The constraints are translated separately. Each of them produces a set of **beforeInsert/Remove** operations, which are all collected together. For simplicity, the sugared notation for **has insert** and **remove** on classes is used.

beforeInsert/Remove constraints

new (**assocOf** *Signature* {**beforeInsert** *expr*}
{**beforeRemove** *expr*}): **Assoc** *Signature*

The expression above is translated as:

```
let self: Assoc Signature = new (assocOf Signature)
in (  self.beforeInsert(fun(Signature) expr);
      self.beforeRemove(fun(Signature) expr);
      self
   )
```

fun(*Signature*) *expr* returns a function whose formal parameters are defined by *Signature*. **let** introduces and binds a new identifier; the form **let** ... **in** *scope* introduces it into the local scope *scope*. *(expr;...;expr)* evaluates the expressions and returns the value of the last one.

All the expressions produced by the successive translations are added in the scope of the **let** above, so that the identifier *self* can be used in all of them.

On classes, **beforeInsert/Remove** are translated in the same way.

key constraints

new (**assocOf** Signature **key** keylist_1 ... **key** keylist_l)

key constraints belong to the kernel language, nevertheless their precise meaning can be defined by a pre-operation, as happens for the derived constraints. Let $l_1{:}T_1,\ldots,l_j{:}T_j$ be a subset of the association signature; then the constraint **key** $l_1,\ldots,l_j$ forces the automatic production of the following pre-operation:

self.**beforeInsert**
(**fun**(Signature) **assert not** self.**has**(**let** $l_1 = l_1 \ldots$, **let** $l_j = l_j$));

Inclusion constraints

new (**assocOf** Signature **are** $A_1,\ldots,A_j$)
new (**classOf** ElType **are** $A_1,\ldots,A_j$)

The above declarations ask the system to maintain automatically an inclusion relation between the new association and each of the immediate superassociations. They are enforced by defining an insertion pre-operation which inserts the element in the immediate superassociations, and a removal pre-operation in any superclass which removes the element from immediate subassociations (the signatures $SignA_1,\ldots,SignA_j$ of $A_1,\ldots,A_j$ are super-signatures of the signature *Sign* of the association defined):

self.**beforeInsert**(**fun**(bind:Sign) C_1.**insert**(bind))
C_1.**beforeRemove**(**fun**(bind:$SignA_1$)
self.**remove**(bind:$SignA_1$))
...
self.**beforeInsert**(**fun**(bind:Sign) C_n.**insert**(bind))
C_n.**beforeRemove**(**fun**(bind:$SignA_n$)
self.**remove**(bind:$SignA_n$))

The insertion messages C_j.**insert**(*bind*) are type correct since *Sign* is a subsignature of $SignA_j$: the type of the argument of **insert** must be a subtype of the type of the association. On the other hand, the messages *self*.**remove**($SignA_j$) are type correct even though $Sign \le SignA_j$, since **remove** accepts arguments belonging to any supertype of the signature of the association. The translation is identical for classes.

An inclusion constraint $R \le S$ only forces a set-inclusion relation between two associations R and S if the corresponding signatures *SignR* and *SignS* are *equality compatible*, i.e. they are associated with the same equality operation. Otherwise, if more bindings, which are all mutually different in *SignR* but equal in *SignS*, are inserted in R, only the first of them is inserted in S, and when this binding is removed from R, all the corresponding bindings are removed from S. In this case this "inclusion" constraint does not model set inclusion exactly but only set inclusion modulo equality, i.e. P.E.R. inclusion as discussed in [Ghelli 90b]. On the other hand, inclusion modulo equality coincides with set inclusion on associations when a key constraint is defined on a component of the superassociation, and on classes when the element type is an object type.

mutual disjointness

new (**classOf** ElType **butnot** $B_1,\ldots,B_n$): **Class** Type

This constraint specifies that the classes $B_1,\ldots,B_n$ must never intersect *self*. If TB_i is the type of the elements of the class B_i, it is only well typed if *ElType* is compatible with TB_i.

This constraint is enforced defining the following insertion preconditions (*beforeIns* stands for **beforeInsert**):

self.**beforeIns**(**fun**(elem:Type) **assert not** (B_1.**has**(elem))
B_1.**beforeIns**(**fun**(elem:TB_1) **assert not** (self.**has**(elem))
...
self.**beforeIns**(**fun**(elem:Type) **assert not** (B_n.**has**(elem))
B_n.**beforeIns**(**fun**(elem:TB_n) **assert not** (self.**has**(elem))

Note that **has** is well typed since TB_i is compatible with *ElType*.

referential constraint

label: *Type* **in/are/owned_by** *class*

In the cases of **in** and **owned_by**, *Type* is compatible with the element type *ElType* of *class*; in the case of **are** *Type* must be a subtype of *ElType*. This is the translation:

label: Type **in** class: **class** ElType →
self.**beforeIns**(**fun**(bind:Sign) **assert** class.**has**(bind.label))
class.**beforeRemove**
(**fun**(el:ElType) **assert not** (self.**has**(**let** label =el)))

```
label: Type owned_by class: class ElType →
self.beforeIns(fun(bind:Sign) assert class.has(bind.label))
class.beforeRemove
        (fun(el:ElType) self.remove(let label=el))
```

```
label: Type are class: class ElType →
self.beforeInsert(fun(bind:Sign) class.insert(bind.label))
class.beforeRemove
        (fun(el:ElType) self.remove(let label=el))
```

Notice that the translation of the **are** referential constraint is identical to the **are** inclusion constraints, justifying the notation.

surjectivity constraints

label: *Type* **onto/owns** *class*

Type is compatible with the element type *ElType* of *class*. This is the translation:

```
label: Type onto class: class(ElType) →
class.beforeInsert(fun(el:ElType) defer assert
  class.has(el) implies self.has(let label=el))
self.beforeRemove(fun(bind:Sign) defer assert
  class.has(bind.label) implies
        self.has(let label=bind.label))
```

```
label: Type owns class: class(ElType) →
class.beforeInsert(fun(el:ElType) defer assert
  class.has(el) implies self.has(let label=el))
self.beforeRemove(fun(bind:Sign) defer
  if self.has(let label=bind.label)   then skip
  else class.remove(bind.label);
```

A **implies** *B* is a boolean expression equivalent to ((**not** *A*) **or** *B*).

constancy constraints

new (assocOf *Signature*
    constant_on label$_1$ **in** class$_1$ label$_n$ **in** class$_n$)

Constancy on a set of components (each associated with a class) means that, once a binding *b* for those components has been fixed, all the bindings extending *b* must be inserted when the elements appearing in *b* are inserted in their classes, and can only be removed when the elements in *b* are removed from their class (at least one of them). This is not the only possible interpretation of the constancy constraint; different interpretations can be enforced procedurally.

The type constraint is that the type of the components must be compatible with the associated classes; the translation is as follows:

```
self.beforeInsert(fun(bind:Sign) assert
  not (    (old class_1).has(bind.label_1)
            and ...
            and (old class_n).has(bind.label_n)
          )
self.beforeRemove(fun(bind:Sign) defer assert
  not ( (class_1).has(bind.label_1) and ...
          and (class_n).has(bind.label_n))
```