1

ROLE-BASED PERSISTENCE

Jürgen Schlegelmilch

University of Rostock,Computer Science Dept., D-18051 Rostock, Germany

ABSTRACT

In most object-oriented systems, references are the only way to express the coupling of objects. They define the visibility among objects, associations of objects, and often lifetime dependencies. We present a relationship construct that separates these three issues, thereby allowing to reduce dependencies and enhance potential parallelism. In this paper, we concentrate on the flexible definition of lifetime dependencies: The persistence of an object is determined by the connections it has with other objects.

1 INTRODUCTION

In object-oriented systems, objects communicate with and depend on other objects. The connections are often modeled by references between objects, whereas in specification models general relationships are used. References define a tight coupling, while relationships can keep the coupling as low as needed, in order to enable inter-object concurrency. One form of coupling is lifetime dependency, or persistence: In many systems, the lifetime of an object depends on that of objects referencing it. In this paper, we present a relationship mechanism with a persistence model that separates the issues of connections, visibility, and lifetime dependencies between objects. This allows the programmer to choose only those dependencies that are needed, thereby reducing cohesion between objects. This work is part of the OSCAR[1] project [10] where we develop an object-oriented database management system; OSCAR is based on the da-

[1] **O**bject Management **S**ystem for **C**omplex **A**pplications, Approach: **R**elational

tabase model EXTREM[2], offering all structural elements of an object-oriented database management system.

Road Map The remainder of this paper is organized as follows: The first part introduces our relationship proposal. Section 2 motivates our decisions in favor of relationships and sketches the idea of the relationship mechanism, while Section 3 defines its syntax and semantics. Section 4 describes operations on relationships, and Section 5 introduces derived relationships. The second part of the paper presents the concept of role-based persistence in Section 6, which is used to model other persistence models in Section 7. Finally, Section 8 summarizes the contributions.

2 MOTIVATION

Many object-oriented specification models offer several relationship types, e.g. aggregation, association, and message connection. Apart from the set of related objects, they describe interaction, integrity constraints, attributes of the connection, and lifetime dependencies. With a relationship construct, we can specify these properties as needed, thereby tuning the amount of cohesion between the related objects. In contrast, the references that most object-oriented programming languages use to connect objects have a fixed set of these properties:

- References connect one object to another, so they are inherently binary.
- They combine connection and visibility: the referenced object is made visible to the referencing one; thus, references are directed.
- In some systems [15, 9, 6, 22], a reference also defines a lifetime dependency: the target object depends on the referencing object.
- No attributes can be given to describe the connection.

References are often stored in instance variables, thus spreading the relationship among several objects and classes. To find a related object, we have to access the state of an object, thereby blocking concurrent accesses to it. A detailed criticism on relationships based on references is given in [19] and [21].

[2] **Ext**ended **Re**lational **M**odel

The Relational Approach The relational model has more capabilities to model relationships. In fact, everything in a relational database is modeled as a relationship between values. Relations can be interpreted to either represent a set of entities, or denote a relationship via foreign key dependencies[3]. By replacing foreign keys with object identifiers, we get a solid base for modeling relationships without sacrificing the advantages of the relational model.

We interpret the tuples in a relation as one–to–one connections between the objects denoted by their object identifiers. Other components of the tuple describe the connection between the objects. With this relational background, we can use standard relational query languages to handle our relationships, and benefit from the research in the relational database area.

In this article we use the algebra for nested relations from [20] with the following syntax (with relations R, S, attributes a, n, lists of attributes L, and predicates *pred*):

operator	syntax	operator	syntax	operator	syntax
instantiation	R	selection	$\sigma[pred](R)$	projection	$\pi[L](R)$
natural join	$R \bowtie S$	renaming	$\beta[n \leftarrow a](R)$	union	$R \cup S$
nesting	$\mu[n; L](R)$	unnesting	$\nu[n](R)$	intersection	$R \cap S$

To relate one object o to n others, we need n tuples in the relation, each tuple describing a one–to–one connection. With nested relations and a suitable algebra, these n tuples can be nested into one with an attribute holding o and another, relation-valued attribute holding unary tuples with the n objects. Since this structure is only one view supporting object o, we prefer the flat relational model for specification.

Bare relations do not provide any kind of access from an object to a related one, nor do they specify lifetime dependencies among them. We therefore add constructs to declare access methods and dependencies, keys and general integrity constraints.

3 THE RELATIONSHIP MECHANISM

The database model EXTREM The EXTREM model supports both values and objects. Values are grouped into types, objects are grouped into classes. The classes are divided into abstract and free ones and placed into an inheritance lattice formed by the subset relation on class extents. Each type and class

[3] Mixed cases are possible and can be seen as an optimization.

describes the structure and behavior of its elements by typed attributes and methods. Each object belongs to exactly one abstract class and may belong to several free classes; it can move into and out of subclasses, and has a set of values for the attributes of the classes it belongs to. Objects of subclasses may be substituted for those of super-classes; for method calls, dynamic binding picks the most specific implementation regardless of the variable the object is bound to. In contrast, values can only be used with their real type, and implementations for their methods are linked statically. Both types and classes have a set *extent* of their values and objects, resp. In the sequel, Types is the set of all types, and Classes the set of all classes.

Relationships In the original EXTREM database model [10], attributes are allowed to have a class type. We replace this kind of attributes by relationships with relationship attributes, and access methods in classes.

Definition (*relationship attribute*)
A relationship attribute is an object a with a name $token(a)$ and a type $type(a) \in \mathsf{Types} \cup \mathsf{Classes}$; *token* has to be globally unique. We call a a role if $type(a)$ is a class type.

Definition (*relationship*)
A relationship is an object R with a set *attrs* of relationship attributes, a set $vital(R) \subseteq attrs(R)$ of roles, a set $extent(R)$, and a set $cons(R)$ of constraints (predicates).

The domain of R is defined as

$$dom(R) = \{c \mid c = \{token(a) : \ \rightarrow extent(type(a)) \mid a \in attrs(R)\}\}$$

i.e. elements of $dom(R)$ are sets of constant functions having the names of the relationship attributes; we call these sets connections. We write $a(c)$ to denote the result of the function a in the connection c. The set $attrs(R)$ is often called the **schema** of R.
Persistence: The set $vital(R) \subseteq attrs(R)$ of roles specifies the roles keeping their objects persistent (see Section 6).
Extent: Although a relationship looks much like a set–of–tuples type, connections have to be created and deleted explicitly like objects. The set $extent(R) \subseteq dom(R)$ holds the connections of R. $c \in extent(R)$ represents a connection between the objects specified by its roles which is further described by the other relationship attributes.
Playing roles: An object o plays the role r in the connection c iff $r(c) = o$; it plays the role in the relationship R iff $\exists c \in extent(R) : r(c) = o$.
Constraints: The set $cons(R)$ of a relationship R is a set of predicates of first

order predicate logic with quantifiers over the schema of R and global names; it restricts the set of connections $extent(R)$. R is called *consistent* if and only if $\forall p \in cond(R) : extent(R) \models p$.

Syntax We use the following syntax to define relationships:[4]

relationship ::=`relationship` *relation-name*
`(` *rel-definition* `)`
{`;` *key-definition*}
[`;` *constraint*]
[`;` *persistence*]
{`;` *access-definition*} `.`
rel-definition ::=*schema-definition* | *query-expression*
schema-definition ::=*col-definition* {`,`*col-definition*}
col-definition ::=*column-name*`:` *type*[*cardinality*]
cardinality ::=`[`*limit*[`:`*limitI*][`,`*limit*[`:`*limitI*]]`]`
limit ::=*Number*
limitI ::=*limit*|`*`
key-definition ::=`key` *column-name* {`,`*column-name*}
constraint ::=`with` *condition*
persistence ::=`vital` *role-name* {`,`*role-name*}
access-definition ::=`in` *role-name*`.` *att-name* `as` `(` *query-expression* `)`

The semantics of these constructs is explained in the sequel using the example in Figure 1. The relationship `family` models a ternary relation between persons. Each `Person` can be the `child` of exactly one (`father`,`mother`) combination, but for some `Persons` we do not know the parents; `Person` is the union of `Female` and `Male`. Each `Male` can be `father` arbitrarily often with different `Female`s, and vice versa with `Female`s and `mother`. Parents access their partners and children via the method `family`, and children access their parent tuple via the method `parents`.

Semantics The *rel-definition* defines the schema of the new relationship R. In the *schema-definition* clause, we define the schema explicitly: Each *col-definition* defines a relationship attribute in $attrs(R)$ with the given name and type. If the name *token* has already been used in another relationship definition, it denotes the same relationship attribute, so the types must be identical.

[4] `Typewriter` font denotes keywords, non-terminal symbols appear in *italics*. Brackets [] enclose optional parts, parts in braces {} may be repeated zero or more times, and the bar | separates alternatives. Concatenation has precedence over |; parentheses ( ) can be used to change precedence.

```
relationship families
    (father:Male[1,0:*],
     mother:Female[1,0:*],
     child:Person[1,0:1]);
   vital father,mother;
   in father.family as
     (π[mother,child](σ[father=this](families)));
   in mother.family as
     (π[father,child](σ[mother=this](families)));
   in child.parents as
     (π[father,mother](σ[child=this](families)));
```

Figure 1 Relationship between classes `Male`, `Female`, and `Person`

Using the *query-expression* instead of the *schema-definition* defines a derived relationship; we discuss this in Section 5.

The *cardinality* clause for an attribute a adds cardinality constraints to the set $cons(R)$. This clause defines one or two intervals of $\mathbf{N} \cup \{\infty\}$, with ∞ denoted as `*`; intervals $n:n$ are abbreviated to n. The first interval is called the inner range and restricts the number of connections that only differ in this role; therefore, its lower bound must not be zero. The second interval is called the outer range and specifies, how often an object may play the role a in this relationship. This range is only needed for roles: multiple occurrences of a value are unrelated to each other, while multiple occurrences of an object share the same state. For the inner range $[n:m]$ and outer range $[n':m']$ of a role a, the following constraints are added to $cons(R)$:

$$\forall t \in \pi[a_i \in attrs(R), a_i \neq a](R) : n \leq \left|\sigma[\textstyle\bigwedge_{a_i \neq a} a_i = a_i(t)](R)\right| \leq m \quad (1.1)$$

$$\forall o \in extent(type(a)) : n' \leq \left|\sigma[a = o](R)\right| \leq m' \quad (1.2)$$

Example In Figure 1, for role `mother` the formulas (1.1) and (1.2) read

$$\forall e \in \pi[\texttt{father}, \texttt{child}](\texttt{families}) :$$
$$1 \leq \left|\sigma[\texttt{father} = \texttt{father}(e) \wedge \texttt{child} = \texttt{child}(e)](\texttt{families})\right| \leq 1$$
$$\forall d \in \texttt{Female} : 0 \leq \left|\sigma[\texttt{mother} = d](\texttt{families})\right| \leq \infty$$

This means, that any combination of `father` and `child` has exactly one `mother`, and a `Female` does not have to be a `mother`, but could be arbitrarily often.

If the upper bound of the outer range of a role is 1, the role is a key for the relationship. The *key-definition* clause allows the definition of additional compound keys. The *constraint* clause takes a predicate of first order predicate logic over the columns of the relationship. Both clauses insert suitable predicates into the set $cons(R)$.

The *persistence* clause inserts the roles in its argument list into the set $vital(R)$. This set defines role-based persistence and is discussed in detail in Section 6.

3.1 Defining Visibility

With a relationship, we can describe connections between objects, but the related objects cannot access each other directly. Therefore we need to define access methods for the roles. These methods are not part of the relationship but of the classes of the roles. The *access-definition* clause defines an access method with name *att-name* in the class of *role*. In their implementation, these methods can e.g. select from the relationship those connections where the current object plays the role in question, thus returning a local view on the relationship. Since objects in (object-preserving) views are fully updatable [12], access methods offer the functionality of references, except for establishing and breaking up a connection. In fact, access methods give more functionality, as they can provide transitive, reflexive, or symmetric closures of relationships as proposed for the ODMG'9X standard in [5]. They also make visibility independend from other properties of the relationship.

Example In Figure 1, three access methods are defined: Both `mother` and `father` have a method `family` returning the combinations of a child and the other parent, and the `children` have a method `parent` computing the tuple of `Female` and `Male` that are her or his parents.

Several connections sharing some objects in a role can be interpreted as a set-valued connection. With nested relational algebra, access methods can restructure the relationship to make such connections visible. By using flat relations, we do not impose a fixed view on the relationship but leave it to the access methods to present structured views.

Example In Figure 1, the access method `family` for role `father` can group the children according to the mother, with the implementation

$$\mu[\texttt{children}; \texttt{child}](\pi[\texttt{mother}, \texttt{child}](\sigma[\texttt{father} = \texttt{this}](\texttt{families})))$$

The result is a relationship with schema {`mother`, `children`}, where `children` is a set of tuples with one component `child`. The extent contains tuples of `mother`s together with the set of `children` she has with the `father` in question.

Related Work Postgres [23] uses a similar approach in a relational environment to provide access between tables that are connected via foreign keys. It allows to use procedures as attributes of a relation. Reading the attribute value returns the result of the procedure.
[19] describes simple access methods to encapsulate insertions, deletions, and selections on relationships. No operators for restructuring relations are provided.
[7] discusses derived attributes for relationships; these attributes are read-only references and cannot offer a restructured view onto the relationship. The scope of roles and attributes of a relationship can be restricted to the participants, so they look like attributes of the objects. The system automatically defines methods to insert and delete connections for the relationship as well as for the derived attributes in the objects.

4 OPERATIONS ON RELATIONSHIPS

Only generic operations are needed to manage the set of connections of a relationship. `INSERT` adds a connection, and `DELETE` deletes a connection specified by a key. Of course, the resulting relationship has to be consistent. Reactions on integrity violations are flexible [21]. Retrieval is done using any of the relational query languages. In our database system OSCAR, all query languages include a complete nested relational base language that we can use to query relationships. To maintain referential integrity, the runtime system propagates deletions of objects to all relationships: if an object o gets deleted, all connections in which o plays a role are deleted from all relationships, too. Otherwise, there would be invalid object identifiers in relationships.

5 DERIVED RELATIONSHIPS

In relational database systems, we have base tables that are managed by explicit `INSERT` and `DELETE` operations, and views that are defined by a query. The relationships we discussed so far correspond to base tables, but our relational approach allows us to define views as well. We call them derived relationships,

and define them using the *query-expression* alternative of the *rel-definition* clause. Both the schema and the extent of a derived relationship, as well as the set of integrity constraints, are defined by the query expression. The defining query can be formulated in any relational query language. These languages are efficiently implamentable, optimizable, and are guaranteed to deliver finite results.

Example An example of a derived relationship is the `parents` relationship, that can be derived from the `families` relationship in Figure 1 like this:

$$\begin{array}{l}\texttt{relationship parents}\\ \quad (\ \beta[\texttt{parent} \leftarrow \texttt{father}](\pi[\texttt{father}, \texttt{child}](\texttt{families}))\\ \quad\ \cup\beta[\texttt{parent} \leftarrow \texttt{mother}](\pi[\texttt{mother}, \texttt{child}](\texttt{families}))) .\end{array}$$

The `parents` relationship is the set of tuples of a parent and one of its children; the schema is {`parent,child`}. This is only possible because the type of the `parent` role can be generalized from `Male` and `Female`, resp., to `Person`.

Discussion Object-oriented query languages like the ODMG'93 standard query language OQL [5] or our object algebra ABRAXAS [10] offer object-generating clauses to be able to represent new combinations of existing objects. If these clauses are evaluated n times, n objects would be generated for the same combination. To avoid these multiple objects, the new object identifiers are usually derived from the combination by a function [14, 11] so that a second evaluation of the object-generating clause yields the same set of objects. So, these object identities are functionally dependent on the state of the object, instead of only determining it. The persistence of the generated objects is also questionable: they strongly depend on the objects they reference. Usually, it is just the other way round: a referenced object depends on the referencing object, not vice versa. With our relationship mechanism, the need for object-generating queries is much weaker, as new combinations of objects can be modeled by derived relationships.
The integration of relationships into OSCAR's query languages is beyond the scope of this paper.

Related Work In [19], [2], and [1], connections are always established and removed by explicit action. [19] only provides operations for membership test, simple selections, and full scans, and leaves it to the programmer to code higher level operations. This set of operations is meant for access methods, not for views. [1] includes a relational-like algebra that is capable of computing new relationships, but the language only allows to define snapshots, not views. So,

none of these approaches supports derived relationships.
[7] does not discuss intensional relationships either, but the underlying language Prolog would be able to provide derived relationships. Connections are modeled as objects, and objects are implemented as sets of facts. Prolog can derive new facts from known ones, thus creating derived connections, but it would then have the problems we mentioned for object-generating queries. However, [7] already admits to basically ignore the object identity of connections.

6 ROLE-BASED PERSISTENCE

Objects have to be created and eventually destroyed. While it is usually clear when to create a new object, it is often unknown when a particular object can be deleted. Most procedural programming languages, e.g. C++ [8], leave it to the programmer to determine this situation and to explicitly delete the object. Others, e.g. Smalltalk [9] and Eiffel [15], employ a garbage collector to delete objects. The garbage collector will delete an object if it is unreachable, i.e. there is no reference to it. Databases can extend the life of an object beyond the end of an application by storing it; we then call the object *persistent*, in contrast to *transient* objects that are deleted at the end of the application. However, not all objects are worth being made persistent, and there are different policies how to determine which objects are, and which are not.

- The programmer explicitly marks objects to be persistent. To get them out of the database, he has to either make them transient again, or delete them. So, persistence is managed by the programmer.
- The programmer describes declaratively, which objects should be persistent. The system will then keep exactly those objects persistent that match the given description, freeing the programmer from the responsibility to manage persistence per object.

Our approach For our database system OSCAR, we follow the second approach. The basic idea is to determine the relevance of an object to other objects: If an object is important for other objects, it will not be deleted before them. This dependency is described by a connection between these objects: Objects playing a role are dependent on the objects playing the other roles in the same connection. We then distinguish ordinary roles from important ones, and call the latter *vital* roles. The *persistence* clause allows to declare a role to be vital; for a relationship R, $vital(R)$ is its set of vital roles.

Definition The definition of role-based persistence is:

> Let Relationships be the set of all relationships. Then the set
>
> $$\bigcup_{R \in \mathsf{Relationships}} \{o \mid \exists c \in \mathit{extent}(R), r \in \mathit{vital}(R) : o = r(c)\}$$
>
> is the set of persistent objects.

A garbage collector only has to mark the objects in this set, and delete all others. To do so, it does not have to access an object's state and can therefore work in parallel with other computations on the objects. Also, relationships are usually much smaller in size than the whole set of objects, thus providing more locality of access.

Example In Figure 1, we have two vital roles, namely `mother` and `father`. Therefore, for any `Person`, the parents are persistent. Since the class `Person` is the union of the classes `Female` and `Male`, the `child` may itself be a `father` or `mother`. Therefore, all ancestors of a persistent person are also persistent. However, without another relationship with a vital role for a person, no object will be persistent at all since the parent–child relationship should be acyclic.

Discussion Role-based persistence is very flexible: If the vital role is the only one in a relationship, an object playing it does not depend on other objects at all. It will be persistent until it is deleted from the relationship. If there are n ordinary roles besides the vital one, objects in this role depend on the other n objects: if any of them is deleted, the connection is also deleted, with the object losing the vital role. Note, that the database system OSCAR also has an explicit `DELETE` command that deletes an object regardless of the roles it plays.

Combining role-based persistence with derived relationships, we achieve full declarative persistence: any query can define a derived relationship, and if we declare its roles to be vital, then all objects in the query result are persistent. Of course, to evaluate the query it is generally required to access the object's state; this increased cohesion is unavoidable if objects are related because of the state they have. Note that being vital is not a property of a role; the relationship defines which roles are vital in its schema. Therefore, roles in a derived relationship are not necessarily vital even if they are in a base relationship.

Related Work We know of no other persistence model comparable in expressiveness with role-based persistence plus derived relationships. The persistence

model in [1] is equivalent to pure role-based persistence. For each relationship, the programmer defines the runtime system's reaction on the deletion of objects and of connections. Alternatives are propagation of deletion to related objects and rollback. While this approach can achieve the same effects as pure role-based persistence, it is not as simple to understand, and less declarative. Without derived relationships, it cannot offer full declarative persistence.
The approach of [2] defines general relationships and a binary, acyclic relationship `has-part` for modeling whole–part relations; this relationship is very close to a reference. The `part` object in this relationship can be declared to be dependent on the `whole` object. It will be deleted with the last `whole` object it is connected to.
[7] distinguishes optional and obligatory relationship attributes. Deleting an object in an optional attribute has no effect on the connection, while for those in obligatory attributes the whole connection is deleted. The underlying language Prolog does garbage collection based on reachability by references from a root set.
Other proposals for relationship mechanisms like [19] do not discuss lifetime dependencies.

[3] demands persistence to be orthogonal to types and transparent to programs; this implies that the persistence model has to be formulated without classes, and it must not require special attributes or methods. Role-based persistence meets both demands: relationships can hold values of any type and objects of any class, and since persistence is realized outside of classes and types, programs handle transient and persistent data transparently.

7 COMPARISON

7.1 Basic Assumptions

We now show for a number of persistence models, how they can be emulated within our framework. The aim is to have the same set of objects persistent at the end of a transaction, given the same sequence of operations. The emulation is done by giving transformation rules to map the constructs of a persistence model to declarations and operations for our database prototype OSCAR. In addition to the mapping, a constant part may be needed to completely cover the model we are emulating. We use C++-like syntax to keep the mapping simple, ignoring any syntactic differences between the programming languages.

We need only few concepts of the OSCAR database system, namely relationships with role-based persistence, and the generic `DELETE` command. Since the declarations of classes are always persistent in OSCAR, there is no need to mark classes for inclusion into a database schema. For simplicity, we assume the existence of a null value NULL although OSCAR has none. The class Object is the top element of the class lattice in OSCAR: any other class is a subclass of Object.

7.2 Replacing References

One–to–One Since almost all programming languages use references instead of relationships, we have to define a replacement for them. If we have a reference named v to an object of class Y in a class X, we replace this declaration with the following relationship:

```
relationship X_to_Y_in_v (aX:X[1:*,1],aY:Y[1,0:*]);
  in X.v as (π[aY](σ[aX=this](X_to_Y_in_v))).
```

The cardinalities ensure that each object in class X is related to one object in class Y, but there may be many X objects related to the same Y object. The access method v already provides read-access to the related object, so we only map assignments to the variable to operations on the relationship:

expression	OSCAR equivalent
`o.v=y;`	`DELETE (aX=o,aY=o.v) FROM X_to_Y_in_v;`
	`INSERT (aX=o,aY=y) INTO X_to_Y_in_v;`

Of course, the `DELETE` and `INSERT` operation have to be performed inside a transaction to make the change atomic and keep the relationship `X_to_Y_in_v` consistent.

One–to–Many References from an object of class X to a set of objects of class Y require only a minor change to the cardinalities of the relationship. For a reference v of type `set(Y)` in class X, we use the following relationship:

```
relationship X_to_Y_in_v (aX:X[1:*,0:*],aY:Y[1:*,0:*]);
  in X.v as (π[aY](σ[aX=this](X_to_Y_in_v))).
```

The mapping of assignment operations is simpler for this kind of reference:

assignment	OSCAR equivalent
o.v+=y;	INSERT (aX=o,aY=y) INTO X_to_Y_in_v;
o.v-=y;	DELETE (aX=o,aY=y) FROM X_to_Y_in_v;

Again, the access method already covers read-access to the related object.

With these mappings, we can now model any kind of reference in the EXTREM data model with relationships. All mappings can be performed mechanically by a preprocessor.

7.3 Persistence by Inheritance

Description In this persistence model, persistence of an object depends on its class membership. A special class defines methods and instance variables to make an object persistent. In the C++ binding of the ODMG'93 standard [5] this class is named `Persistent_Object`. A class is called persistent if it inherits from this special class, and only objects of persistent classes can be persistent. An object of a persistent class is persistent if it is assigned to a database, and transient otherwise. This assignment is done by a parameter *db* to the operator `new` specifying a database that the new object has to be assigned to. The method `delete_object()` of class `d_Ref` deletes the object bound to a reference from both memory and database. While the first version of the ODMG'93 standard [4, p.19] defined three lifetime models, the revised version [5] only distinguishes transient and persistent objects.

Implementations The C++-based ODBMS Poet [18] closely follows the idea of the ODMG'93 persistence model. In Poet's native API, the special class is called `PtObject`, and the assignment to a database *db* is done with the method `Assign(`*db*`)` of that class. Unassigning an object with the method `UnAssign()` of class `PtObject` makes it transient. However, the object is not automatically deleted as required by the ODMG'93 standard; this must be done manually with C++'s operator `delete`.

The ODBMS O_2 [6] claims to be ODMG-compliant, but its C++ interface [16] defines the class `Persistent_Object` with no instance variables, and all methods of this class including `delete_object()` do nothing. Persistence in the O_2 system is defined by reachability (see Section 7.4); assigning or unassigning an object has no effect on its persistence. Thus, O_2 follows the persistence model of ODMG'93 only syntactically.

Mapping To model the ODMG'93 persistence model, we provide a class and a relationship in the constant part of the mapping:

```
class Persistent_Object {};
relationship is_Persistent (theObject:Persistent_Object);
  vital theObject.
```

Like O_2, we do not really need a special class for persistence. We now map functions of the C++ binding of ODMG'93 to operations on the relationship `is_Persistent`:

ODMG'93 function call	OSCAR equivalent
v=new(*db*) *Class*(...);	v=new *Class*(...); INSERT (theObject=*v*) INTO is_Persistent;
o->delete_object();	DELETE *o*;

Thus, to make an object persistent, we simply insert it into the relationship `is_Persistent`; the role `theObject` is vital and will therefore prevent the object from being deleted by the garbage collector. The `DELETE` command will delete the object in spite of this role.

7.4 Persistence by Reachability

Description This persistence model is used in the ODBMSs O_2 [6], GemStone [22], and in the programming language Eiffel [15]. It is a declarative persistence model based on references. The set of persistent objects is defined as follows:

> Let $ref \subseteq \texttt{Object} \times \texttt{Object}$ be the reference relation, defined by $(o, s) \in ref \iff o$ references s, and ref^* its transitive closure. Then the set of persistent objects is
>
> $$\{o \mid \exists r \in root : (r, o) \in ref^*\}$$
>
> for a set *root* of initially persistent objects.

In O_2, the set *root* is the set of objects bound to persistent variables called `names`. In GemStone, it is the standard dictionary `Smalltalk`, and in Eiffel, it is formed by all objects bound to variables on the stack or in the data segment.

Mapping To achieve persistence by reachability in our persistence model, we replace any reference by a binary relationship as described in Section 7.2, and add the declaration

```
vital a$Y$;
```

to each relationship; this makes the role of the referenced object vital. An object will play the vital role until the connection is deleted from the relationship. This either happens on assignment due to the mapping defined in Section 7.2, or automatically on deletion of the referencing object. The relation *ref* in the definition of persistence by reachability is the union of all the relationships we get as replacements for references.

For the root set, we only show how to model the approach of the ODBMS O_2. The idea is to have a relationship with one vital role and a relationship attribute to hold the name of the persistent variable.

```
relationship root_set (name:String,theObject:Object[1,0:*]);
  vital theObject.
```

The cardinalities imply that `name` is a key for the relationship; to include set-valued names, we would have to change them as shown in Section 7.2 for the mapping of set-valued references, or create a second relationship with the adapted cardinalities.

We then map access operations of O_2 to operations on this relationship:

O_2 operation	OSCAR equivalent
`create name` v	`INSERT (name=`v`,theObject=NULL) INTO root_set;`
`delete name` v	`DELETE (name=`v`) FROM root_set;`
v	π[`theObject`](σ[`name` =`v`]($root\_set$))
v=y;	`DELETE (name=`v`) FROM root_set;`
	`INSERT (name=`v`,theObject=`y`) INTO root_set;`

Creation and deletion of a name are mapped to insertion and deletion of a connection in the relationship `root_set`. Read-access to a name v is mapped to a query on `root_set`; note that we have to cast the resulting object to the desired type. Assignment of a new object to a name is handled similar to assignment to a reference in Section 7.2.

7.5 Persistence by Creation

Description This is the approach taken by ObjectStore [17]: An object is persistent if it is created in persistent memory. There are no restrictions on classes or types. In contrast to the ODMG'93 approach, inheritance from a special class is no prerequisite for persistence.

Syntactically, ObjectStore strongly resembles the persistence-related part of the C++ binding of ODMG'93. The operator `new` has an additional argument called *placement* that determines where to place the new object. If the placement is a database, the object will be created in the database and therefore be persistent. Unlike Poet (see Section 7.3), an object cannot become persistent after its creation, and has to be persistent until it is deleted. Persistence by creation is therefore rather inflexible.

Mapping To model this persistence model in our approach, we use a mapping very similar to that presented in Section 7.3. The class `Persistent_Object` was only necessary for syntactical compliance with the ODMG'93 standard, so we can discard it safely. The constant part of the mapping therefore consists of only one relationship:

```
relationship is_Persistent (theObject:Object);
  vital theObject.
```

The operations of ObjectStore differ from the ODMG C++ binding only in one point: instead of the method `Persistent_Object::delete_object()`, the standard C++ operator `delete` has to be used to delete objects.

ObjectStore operation	OSCAR equivalent
v=new(*db*) *Class*(...);	v=new *Class*(...); INSERT (theObject=*v*) INTO is_Persistent;
delete *o*;	DELETE *o*;

To make persistent objects accessible, ObjectStore offers persistent variables similar to that in O_2. However, since persistence by creation is independent of references, persistent objects can become unreachable. ObjectStore offers no tool to remove such objects from the database. We note that ObjectStore can make values persistent.

7.6 Persistence on Request

Description In the database programming language GOM [13], an object is persistent if it has been sent the message `persistent`. GOM requires the programmer to mark classes with the keyword `persistent` to make the type information persistent; this is not needed in OSCAR.

Mapping The mapping to role-based persistence is simple. The constant part is the same as for the policy presented in Section 7.5:

```
relationship is_Persistent (theObject:Object);
  vital theObject.
```

The method call `persistent` to an object is mapped to inserting the object into the relationship:

GOM operation	OSCAR equivalent
o`->persistent;`	`INSERT (theObject=`*o*`) INTO is_Persistent;`
o`->transient;`	`DELETE (theObject=`*o*`) FROM is_Persistent;`

Note that in GOM there is no way to make a persistent object transient: method `transient` above is a possible extension. With our approach, this can be achieved by deleting the object from the relationship `is_Persistent`.

In GOM, variables can be marked to be `persistent` but this only means that their declaration is persistent; they are not entry points as described in Section 7.4, i.e. they do not make the referenced objects persistent.

8 CONCLUSION

In this paper, we introduced general relationships as a flexible way to describe cohesion between objects. Both visibility and lifetime dependency can be added separately, allowing to keep coupling between objects at a minimum. Our persistence model has been shown to be at least as capable as the persistence models found in other systems, even without taking advantage of derived relationships. Our approach combines the following achievements:

1. The properties of connection, visibility, and persistence are separated from each other. References inherently combine connection with visibility, and persistence by reachability combines all three properties.
2. The concept of derived relationships helps reducing redundancy and avoiding inconsistencies. It also adds expressive power to our persistence model.
3. Role-based persistence decouples objects. Objects can depend on other objects without being made visible, and their lifetime no longer depends on the state of other objects.

Besides integrating our relationship construct into the database prototype OSCAR, we plan to implement it on top of the commercial ODBMS O_2, since its O_2SQL query language supports derived relationships. However, the persistence concept will not be implementable in O_2: connections will be implemented as tuples of references, so related objects will be kept persistent by the connections they are in. O_2 does not have references without persistence.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments improving the quality of this paper.

REFERENCES

[1] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, September 1991.

[2] W. Andreas and T. Gorchs. Relationship service. Technical report, Object Management Group, 1993. OMG TC Document 93.11.9.

[3] Malcolm P. Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *VLDB journal*, 4(3), July 1995.

[4] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, San Mateo, CA, 1994.

[5] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93, version 1.2.* Morgan-Kaufmann, San Mateo, CA, 1996.

[6] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[7] Oscar Diaz and P. M. D. Gray. Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Databases. In *Conf. on Object-Oriented Databases*, Windermere, July 1990.

[8] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[9] A. Goldberg and D. Robson. *Smalltalk 80: The language and its implementation.* Addison-Wesley, 1983.

[10] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. of the 9th Int. Conf. on Entity-Relationship Approach, Lausanne*, pages 95–110. Elsevier, October 1990.

[11] A. Heuer and P. Sander. The LIVING IN A LATTICE rule language. *Data and Knowledge Engineering*, 9(3):249–286, 1993.

[12] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, pages 178–197. Springer, Informatik-Fachbericht 270, 1991.

[13] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM — a strongly typed, persistent object model with polymorphism. In *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, pages 198–217. Springer, Informatik-Fachbericht 270, 1991.

[14] M. Kifer and G. Lausen. F-Logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 134–146. ACM New York, May 1989.

[15] Bertrand Meyer. *Eiffel: The Language.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, 1993.

[16] O_2 Technology. C++ *Interface to* O_2, March 1995.

[17] Object Design Inc. *ObjectStore* C++ *API User Guide*, June 1995.

[18] Poet Software GmbH. *Poet – Programmer's & Reference Guide*, 1994.

[19] James Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 466–481, 1987.

[20] H.-J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information systems*, 11(2):137–147, June 1986.

[21] Jürgen Schlegelmilch. An Advanced Relationship Mechanism for Object-Oriented Databases. Technical Report 19/1996, University of Rostock, Computer Science Dept., 1996.

[22] Servio Logic Development Corp. *GemStone Product Overview*, 1991.

[23] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.