ADDING MORE SUPPORT FOR ASSOCIATIONS TO THE ODMG OBJECT MODEL

Bryon K. Ehlmann
Department of Computer Science, Southern Illinois University Edwardsville, Edwardsville, IL 62026, USA

Keywords: ODMG Object Model, OODB systems, Constraint management, Object Relationship Notation (ORN).

Abstract: The Object Model defined in the ODMG standard for object data management systems (ODMSs) provides referential integrity support for one-to-one, one-to-many, and many-to-many associations. It does not, however, provide support that enforces the multiplicities often specified for such associations in UML class diagrams, nor does it provide the same level of support for associations that is provided in relational systems via the SQL references clause. The Object Relationship Notation (ORN) is a declarative scheme that provides for the specification of enhanced association semantics. These semantics include multiplicities and are more powerful than those provided by the SQL references clause. This paper describes how ORN can be added to the ODMG Object Model and discusses algorithms that can be used to support ORN association semantics in an ODMG-compliant ODMS. The benefits of such support are improved productivity in developing object database systems and increased system reliability.

1 INTRODUCTION

An *object data management system* (ODMS) allows objects created and manipulated in an object-oriented programming language to be made persistent and provides traditional database capabilities like concurrency control and recovery to manage access to these objects. An *object database management system* (ODBMS), one type of ODMS, stores the objects directly in an *object database*. An *object-to-database mapping* (ODM), another type of ODMS, stores the objects in another database system representation, usually relational (Cattel *et al.*, 2000).

The de facto standard for ODMSs is ODMG 3.0 (Cattel *et al.*, 2000), which was defined by the Object Data Management Group (ODMG) consisting of representatives from most of the major ODMS vendors. This standard defines an Object Model to be supported by ODMG-compliant ODMSs. The model defines the kinds of object semantics that can be specified to an ODMS. These semantics deal with how objects can be named and identified and the properties and behavior of objects. They also deal with how objects can relate to one another, which is the focus of this paper.

In addition to supporting generalization-specialization relationships, the Object Model supports one-to-one, one-to-many, and many-to-many binary relationships between object types. These are the non-inheritance, or structural, types of relationships, which are termed *associations* in the Unified Modeling Language (UML) (OMG, 2005), . For example, a one-to-many association between carpools and employees can be defined in the Object Model. A carpool object is defined so that it can reference many employee objects, and an employee object is defined so that it can reference at most one carpool.

The Object Model prescribes that the ODMS automatically enforce referential integrity for all defined associations. This means that if an object is deleted, all references to that object that maintain associations involving that object must also be deleted. This ensures that there are no such references in the database that lead to nonexistent objects.

What has just been described is the extent of support for associations in the Object Model. What is lacking is some additional, easily implementable support for associations that could significantly improve the productivity of developing object database systems and the reliability of those systems.

For example, the Object Model, like the relational model, does not support the specification of precise *multiplicities*. Such association constraints are almost always present in the diagrams used to model databases—the traditional Entity-Relationship

Diagram (ERD) (Chen, 1976), where multiplicities are termed *cardinality constraints*, and the UML class diagram (OMG, 2005). For example, the multiplicity for the Employee class in the carpool–employee association may be given as 2..* in a class diagram, meaning that a carpool must be related to two or more employees. Such association semantics, documented during conceptual database design, are sometimes lost during logical database design unless supported by the logical data model, e.g., the Object Model. If not supported, to survive, they must be resurrected by the programmer during implementation and for object databases translated into cardinality checks on collections and into exception handling code within relevant create and update methods.

The Object Model also does not support association semantics that are equivalent to those supported in standard relational systems via the references…on delete clause of the create table statement in SQL (ANSI, 2003). Such semantics would, for instance, allow one to declare an association between objects such that if an object is deleted, all related objects would be automatically deleted by the ODMS, i.e., an on delete cascade. For example, if an organization in a company were deleted, all subordinate organizations would be implicitly deleted. Such an association semantic is required for an ODMS to provide support for *composite objects*.

Object Relationship Notation (ORN) was developed to allow these kinds of semantics, and others often relevant to associations, to be better modeled and more easily implemented in a DBMS (Ehlmann *et al.*, 1996, 2000, 2002). ORN is a declarative scheme for describing association semantics that is based on UML multiplicities.

In this paper we give a brief overview of ORN and show how the ODMG Object Model can be extended to include ORN. We also discuss and illustrate algorithms that are available and can be used by an ODMG-compliant ODMS to implement the association semantics as specified by ORN. The extension is very straightforward, and the algorithms are relatively simple. The end-result is an enhanced Object Model that supports more powerful association semantics—in fact, more powerful than those supported by relational systems without having to code complex constraints and triggers (Ehlmann and Riccardi, 1996). By extending models with ORN and providing the required mappings between them—UML class diagram to Object Model to ODMS implementation—we facilitate a model-driven development approach and gain its many advantages (Mellor *et al.*, 2003).

The specific benefits here are a significant improvement in the productivity of developing object database applications and an increase in their reliability. Productivity is improved when translations from class diagram models into object models are more direct and when programmers do not have to develop code to implement association semantics. Currently, many developers working on many database applications must implement, test, and maintain custom code for each type of association, often "reinventing the wheel." Reliability is increased when the ODMS is responsible for enforcing association semantics. Currently, developers sometimes fail to enforce these semantics or inevitably introduce errors into database applications when they do.

The remainder of this paper is organized as follows: section 2 gives a brief overview of ORN and related work, section 3 shows how the ODMG Object Model can be extended with the ORN syntax and describes ORN semantics in terms of this model, section 4 discusses and illustrates algorithms that can be used to implement ORN semantics in an ODMS that is based on the extended Object Model, and section 5 provides concluding remarks. A complete set of ORN-implementing algorithms is available on the author's website (Ehlmann, 2006).

2 ORN AND RELATED WORK

ORN describes association semantics at both the conceptual, i.e., data modeling, and logical, i.e., data definition, levels of database development, and can be compared to other declarative schemes.

For data modeling, ORN has been integrated into ERDs and UML class diagrams (Ehlmann and Yu, 2002). ORN extends a class diagram by allowing binding symbols to be given with multiplicity notations. The bindings indicate what should happen when links between related objects are destroyed, either implicitly because of object deletions or explicitly. They indicate, for instance, what action the DBMS should take when destroying a link would violate the multiplicity at one end of an association. The binding symbols (or the lack of them) provide important semantics about the relative strength of linkage between related objects and define the scope of *complex objects*. For example, the association between a carpool, a complex object, and its riders can be specified in an ORN-extended class diagram to indicate that if the number of riders falls below two, either because an employee leaves the company (an employee object is deleted) or just leaves the carpool (a link between an employee and a carpool is destroyed), the carpool should be deleted.

For database definition, ORN has been implemented within the Object Database Definition Language (ODDL). ODDL is a language used to define classes, attributes, and relationships to a prototype ODMS named Object Relater *Plus* (OR+) (Ehlmann and Riccardi, 1997). OR+ closely parallels ODMG and is built on top of Object Store (Progress Software, 2006). The integration of ORN into ODDL allows a direct translation of association semantics from an ORN-extended class diagram into the database definition language and enables these semantics to be automatically maintained by the DBMS. Using ORN, the semantics for an association between employees and carpools as previously described can be both modeled and implemented by the notation |~X~<2..*-to-0..1>. No programming is needed.

In Ehlmann and Riccardi (1996), the power of ORN in describing association semantics is compared to that of other declarative notations proposed for various object models and to that of the references clause of SQL. The comparison reveals that the most unique aspect of ORN, and what accounts for its ability to specify a larger variety of association types, is that it provides for the enforcement of upper and lower bound multiplicities and allows delete propagation to be based on these multiplicities. It also provides a declarative scheme at a conceptual level of abstraction that is independent of database type, object or relational. ORN can also be compared to extensions to the ER model that others have suggested to specify or enforce association semantics, or *structural integrity constraints* (Balaban and Shoval, 2002) (Bouzeghoub and Metais, 1991) (Lazarivic and Misic, 1991). These extensions, however, are more procedural in nature.

3 ADDING ORN TO ODL

3.1 Associations in ODL

In the ODMG Object Definition Language (ODL), which defines the ODMG Object Model, an association is defined by declaring a relationship *traversal path* for each end of the association. A traversal path provides a means for an object of one class to reference and access the related objects of a *target class* (which is the same class in a recursive relationship). Access to many target class objects requires the traversal path declaration to include an appropriate collection type, usually a set or list, that can contain

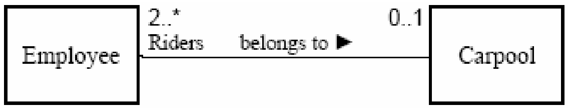


Figure 1: Class diagram for employee–carpool association.

```
class Employee {
   ...
   relationship Carpool          carpool
                                 inverse Carpool::riders;
   ...
};
class Carpool {

   relationship set<Employee>  riders
                                 inverse Employee::carpool;
   ...
};
```

Figure 2: ODL for employee–carpool association.

references of target class type. Access to at most one target class object requires the declaration to include a reference of target class type. A traversal path declaration must also include the name of its inverse traversal path. For example, the one-to-many relationship between carpools and employees, discussed earlier and modeled by the class diagram in Fig. 1, would be declared in ODL as shown in Fig. 2. The 2..* multiplicity given in the class diagram must be implemented by application code.

3.2 Adding ORN Syntax

Adding ORN to the Object Model is relatively straightforward. Essentially, ODL is extended to allow an *<association>* to be given for each declared relationship. The syntax for an *<association>*, which is the syntax for ORN, is given in Fig. 3, and the ORN-extended ODL syntax is given in Fig. 4.

To illustrate the syntax and semantics of ORN in the context of the Object Model, a database containing the employee–carpool association as well as two other associations is modeled by the ORN-extended class diagram given in Fig. 5. In such a diagram, the ORN bindings for a class (or role) in an association are given as stereotype icons at the association end corresponding to that class (or role). When no binding symbols are given for an association end (or role), default bindings are assumed, the semantics of which will be defined later.

The database modeled in Fig. 5 is implemented by the ORN-extended ODL given in Fig. 6.

If an *<association>* is not given for a relationship in ODL (see Fig. 4), the default *<association>* is <0..1-to-0..1> for a one-to-one relationship, <0..1-to-*> for a one-to-many, and <*-to-*> for a many-to-

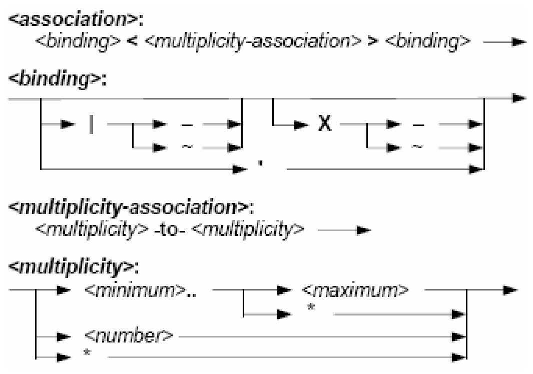
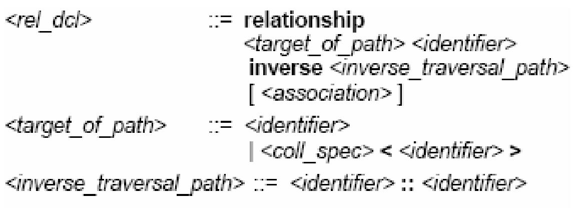


Figure 3: ORN syntax diagrams.



```
<rel_dcl>              ::= relationship
                            <target_of_path> <identifier>
                            inverse <inverse_traversal_path>
                            [ <association> ]
<target_of_path>       ::= <identifier>
                            | <coll_spec> < <identifier> >
<inverse_traversal_path> ::= <identifier> :: <identifier>
```

Figure 4: Updated BNF for a relationship in ODL.

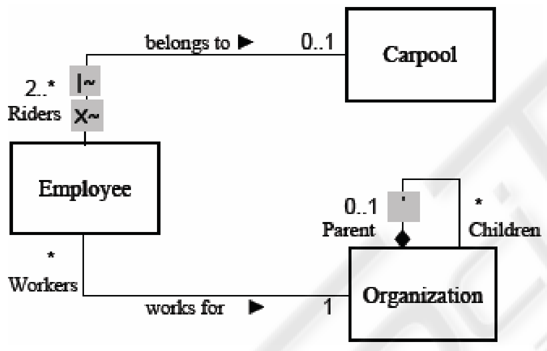
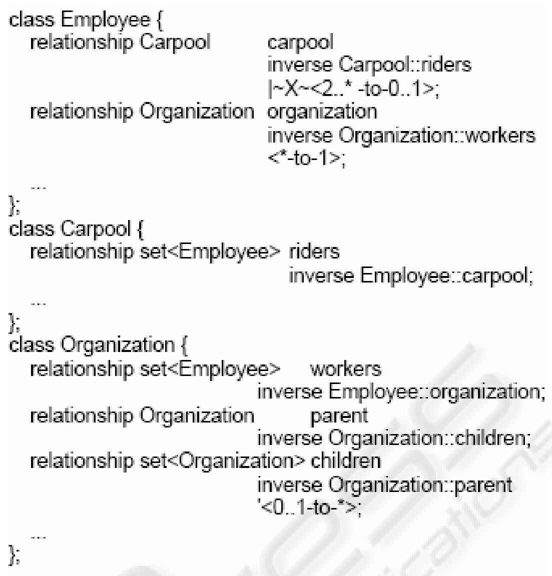


Figure 5: ORN-extended UML class diagram.

many. These defaults give relationships the same semantics as they have in the existing Object Model.

An <*association*> given for a relationship need only to be given for one of the traversal paths. If given for both traversal paths, the <*association*>s must be inverses of each other. For example, an <*association*>, if given for riders in Fig. 6, must be given as <0..1-to-2..*> |~X~.

When an <*association*> is given for a traversal path *tp* in class *C*, the multiplicity and binding given after the -to- apply to *tp* and to the target class, the multiplicity and binding given before the -to- apply to the inverse *tp* and to class *C*. For example, in Fig. 6, the multiplicity 0..1 and default bindings apply to the traversal path carpool and the target class Carpool, and the multiplicity 2..* and binding |~X~ apply to the traversal path riders and class Employee. If the



```
class Employee {
   relationship Carpool        carpool
                               inverse Carpool::riders
                               |~X~<2..* -to-0..1>;
   relationship Organization   organization
                               inverse Organization::workers
                               <*-to-1>;
   ...
};
class Carpool {
   relationship set<Employee>  riders
                                  inverse Employee::carpool;
   ...
};
class Organization {
   relationship set<Employee>     workers
                               inverse Employee::organization;
   relationship Organization      parent
                               inverse Organization::children;
   relationship set<Organization> children
                               inverse Organization::parent
                               '<0..1-to-*>;
   ...
};
```

Figure 6: ODL for class diagram shown in Fig. 5.

multiplicity given for a traversal path in an <*association*> implies "many," then the type of that traversal path must be a collection.

The last issue to address in extending ODL is association inheritance. In the Object Model, a relationship can be inherited by a class via the extends relationship. For example, the declaration **class** SalesPerson **extends** Employee **{** ... **}** would mean that the SalesPerson class inherits the attributes, relationships, and behavior of the Employee class. Thus, the carpool traversal path as declared in the Employee class in Fig. 6 would be inherited by the SalesPerson class, allowing sales people to join carpools. When a relationship is inherited by a class, all of the semantics defined by its <*association*>, given or defaulted, are also inherited.

And, of course, the semantics of all <*association*>s defined in the ODL—defaulted, given, or inherited—must be maintained by the ODMS.

3.3 ORN Semantics in ODL Context

The semantics of the <*multiplicity*>s in an <*association*> are identical to those of the multiplicities defined in UML (OMG, 2005). The semantics of the <*binding*>s are given in Table 1.

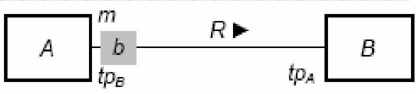
Previous papers have described ORN semantics conceptually in terms of ER and class diagrams, e.g. Ehlmann *et al.* (2002). The reader may review these papers for a more detailed discussion of ORN. Here, we focus more on describing ORN semantics in terms of the Object Model, or ODL. Thus, instead of

Table 1: ORN binding semantics for the Object Model.

Binding semantics are described below in terms of a subject class *A* in a relationship *R* having multiplicity *m* and the given symbol in its binding *b*. The first and second nils listed below denote that *b* contains no implicit and no explicit binding symbols, respectively. *R* relates objects in class *A* to objects in a related, or target, class *B*.

R is implemented by a traversal path *A*::tp_A and inverse traversal path *B*::tp_B. The type of tp_A is *B* or some suitable collection of *B* (shown here as *B*), and the type of tp_B is *A* or some suitable collection of *A* (shown here as set<*A*>). The binding for traversal path tp_B is *b* and the multiplicity is *m*.

The deletion of a *A* object succeeds only if all existing relationship references involving that object are implicitly destructible, i.e., can be dropped. Also, the deletion of an *A* object or the dropping of a tp_A reference succeeds only if all required implicit deletions succeed. Dropping a traversal path reference is equivalent to and includes dropping its inverse traversal path reference.



```
class A {
   ...
   relationship B tpA inverse B::tpB
                         b<m-to-...;
}
class B {
   ...
   relationship set<A> tpB inverse A::tpA
                              ...-to-m>b;
}
```

nil	*Default implicit destructibility*. On delete of an *A* object, a tp_A reference can be implicitly dropped provided this does not violate *m*.*
|–	*Minus implicit destructibility*. On delete of an *A* object, a tp_A reference can never be implicitly dropped. *Implicit* dropping of a reference is denoted by the |, symbolizing a *cut* in the reference.
|~	*Propagate implicit destructibility*. On delete of an *A* object, a tp_A reference can always be implicitly dropped. The target object is implicitly deleted when the dropping of this reference violates *m*.
nil	*Default explicit destructibility*. A tp_A reference can be explicitly dropped provided this does not violate *m*.*
X–	*Minus explicit destructibility*. A tp_A reference can never be explicitly dropped. *Explicit* dropping of a reference is denoted by the X.
X~	*Propagate explicit destructibility*. A tp_A reference can always be explicitly dropped. The target object is implicitly deleted when the dropping of this reference violates *m*.
'	*Prime implicit and explicit destructibility*. On delete of an *A* object, a tp_A reference can be implicitly dropped. Also, a tp_A reference can be explicitly dropped. After the reference is dropped, an implicit delete is attempted on the target, i.e., subordinate, object. This deletion is required, and thus must succeed if and only if its failure and resultant undoing would violate *m*.*

* - The check for a violation caused by dropping the reference is deferred until the end of the current complex object operation.

"association links" being conceptually "created" and "destroyed," "relationship references" (or, alternatively, "traversal path references") are "formed" and "dropped." Dropping a relationship or traversal path reference also means dropping the corresponding inverse reference (in the inverse traversal path). Also, bindings and multiplicities are now associated with traversal paths as well as with the related classes. This is convenient for identifying bindings and multiplicities in recursive relationships since the subject class and related class, now called the "target class," are the same. Traverse path names can be equated to role names given in UML class diagrams. In Table 1, traversal path names tp_A and tp_B are also role names in the class diagram for relationship *R*.

As indicated in Table 1, association semantics are derived from multiplicity semantics and the semantics of the given bindings. For example, in the |~X~<2..*-to-0..1> association between employees and carpools, the |~ symbol in the <*binding*> for the Employee class means (from Table 1): on delete of an Employee object, a carpool reference (see Fig. 6) can always be implicitly dropped, and the target Carpool object is implicitly deleted when dropping this reference violates the multiplicity 2..*. The X~ symbol means: a carpool reference can always be explicitly dropped, and the target Carpool object is implicitly deleted when dropping this reference violates the multiplicity 2..*. The multiplicity 2..* is violated when a reference to one of just two employees in a carpool, i.e., one of just two references in the set riders, is dropped. The default <*binding*> for the Carpool class means (again, from Table 1): on delete of a Carpool object, a reference in riders (see Fig. 6) can be implicitly dropped provided this does not violate the multiplicity 0..1, and a reference in riders can be explicitly dropped provided this does not violate the multiplicity 0..1. A 0..1 multiplicity is never violated by dropping a reference in riders (or a carpool reference for that matter).

Below are more of the association semantics that are modeled in Fig. 5 and implemented in Fig. 6. They are described both conceptually and, within brackets, in terms of Object Model.

- If an employee [Employee object] is deleted, the link to the employee's organization is implicitly destroyed [the object's organization reference to its target Organization object is implicitly dropped] (default binding and * multiplicity).
- If an organization [Organization object] is deleted, all descendant organizations [Organization objects recursively referenced via children] are implicitly deleted (' binding); however, an organization is not deleted if it has any employees [if workers references any Employee objects] (default binding and 1 multiplicity).
- If a link between organizations is destroyed [if a children reference (or its inverse parent reference) is dropped], the child organization and all descendant organizations [Organization objects recursively referenced via children] are implicitly deleted (' binding); however, again, an organization is not deleted if it has any employees (default binding and 1 multiplicity).

4 IMPLEMENTING ORN

The implementation of ORN semantics in an ODMG-compliant ODMS is described by giving the algorithms required to create and delete objects and form and drop relationship references. These operations become complex object operations in the context of ORN. This means they may no longer involve just one object or relationship reference but may involve many objects, relationships, and relationship references in the scope of a complex object.

In Ehlmann (2006), we give the algorithms for these operations by providing all related pseudocode, with commentary, for the ObjectFactory::new() and Object::delete() methods, which are associated with an object, and the *C*::form_*tp*() and *C*::drop_*tp*() methods, which are associated with a declared traversal path *tp* in a user-declared class *C*. These methods are defined as part of the Object Model (see Chapter 2 of Cattel and Douglas(2000)).

In this section, due to space constraints, we discuss these algorithms in general and illustrate them by giving the algorithm for just the Object::delete() method. The pseudocode shown in this section is about one quarter of that given in Ehlmann (2006).

The algorithms have been developed by reverse engineering the code for implementing ORN within OR+. This is the same code executed when one uses the ORN Simulation, a web-based, prototype modeling tool (Ehlmann, 2000). Thus, the algorithms are well-tested but have a slightly different wrapping.

Their implementation of ORN semantics is unambiguous in the presence of association cycles as long as *<association>*s do not contain a |– binding for just one end of the association. By unambiguous, we mean that the results of a complex object operation are independent of the order in which traversal paths and the references in these paths are processed. This property of ORN is discussed in detail and proven in Ehlmann *et al.* (2002).

As stated in the introduction, the algorithms are relatively simple; however, they depend on the ODMS implementation supporting a nested transaction capability. Nested transactions are needed to implement the semantics of the ' (prime) binding and are desirable so that the system can check multiplicity violations at the end of a complex object operation, undoing the operation upon any exception and thus making the complex object operation atomic. The Object Model defines a Transaction Model, which does not provide nested transactions. So, before giving the algorithms for the complex object operations in Ehlmann (2006), we extend the Transaction Model to support nested transactions, at least for the purpose of implementing the ODMS. We assume such support for nested transactions and give algorithms for transaction methods, focusing on the actions required to support ORN semantics.

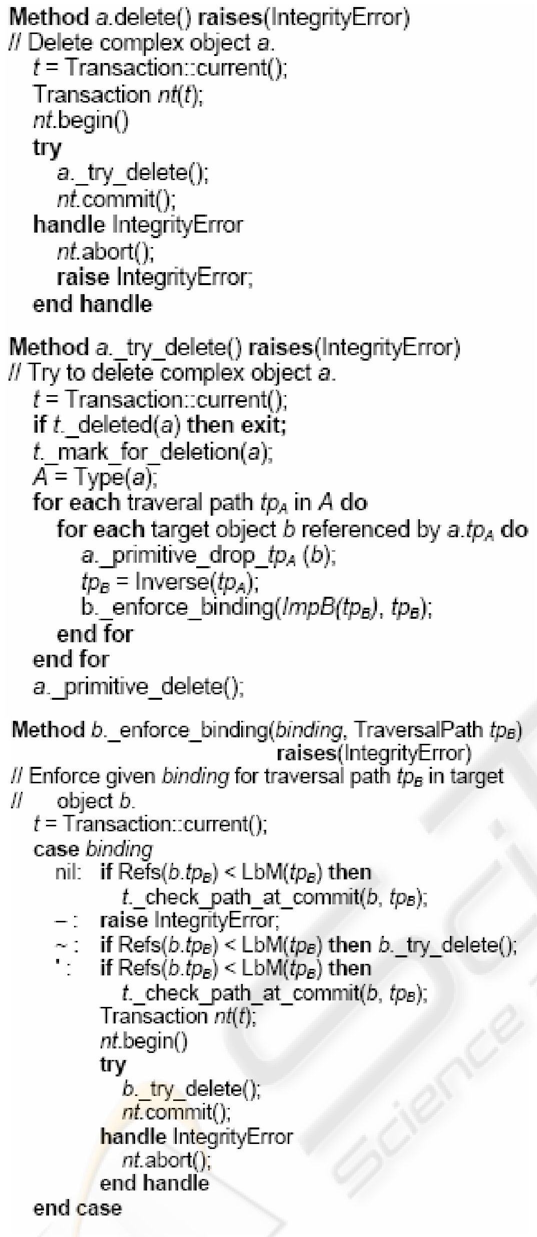
All methods are assumed to execute in the context of a opened database *d*, and methods new(), delete(), form_tp_A(), and drop_tp_A() are assumed to execute within the scope of a user-defined transaction.

The pseudocode that expresses the algorithms is some mixture of ODL, C++, Java, and English. We have tried to stick as close as possible to the conventions of ODL. Indention indicates control structure, with appropriate end's often used to terminate compound statements. The try...handle...end handle control structure for exception handling is similar to Java's try {...} catch {...}. Methods for a class are introduced with a header of the form Method *<variable>*.*<method name>*(...), where the *<variable>* is used in the body of the method to refer to the object on which the method is invoked, i.e., the implicit parameter and this object in C++ and Java. A *<method name>* begins with an underscore if it is to be invoked only by the ODMS implementation.

The algorithms are expressed using the variables defined in Table 1.

Fig. 7 gives the delete() method and two methods that it uses, _try_delete() and _enforce_binding(). The given delete() replaces the primitive delete() method as currently defined in the Object Model.

The remainder of this section briefly explains the pseudocode in Fig. 7. For a more detailed explanation and for the pseudocode of all methods invoked by the delete() algorithm, see Ehlmann (2006).



```
Method a.delete() raises(IntegrityError)
// Delete complex object a.
   t = Transaction::current();
   Transaction nt(t);
   nt.begin()
   try
      a._try_delete();
      nt.commit();
   handle IntegrityError
      nt.abort();
      raise IntegrityError;
   end handle

Method a._try_delete() raises(IntegrityError)
// Try to delete complex object a.
   t = Transaction::current();
   if t._deleted(a) then exit;
   t._mark_for_deletion(a);
   A = Type(a);
   for each traveral path tp_A in A do
      for each target object b referenced by a.tp_A do
         a._primitive_drop_tp_A (b);
         tp_B = Inverse(tp_A);
         b._enforce_binding(ImpB(tp_B), tp_B);
      end for
   end for
   a._primitive_delete();

Method b._enforce_binding(binding, TraversalPath tp_B)
                                  raises(IntegrityError)
// Enforce given binding for traversal path tp_B in target
//     object b.
   t = Transaction::current();
   case binding
      nil:  if Refs(b.tp_B) < LbM(tp_B) then
               t._check_path_at_commit(b, tp_B);
      – :   raise IntegrityError;
      ~ :   if Refs(b.tp_B) < LbM(tp_B) then b._try_delete();
      ' :   if Refs(b.tp_B) < LbM(tp_B) then
               t._check_path_at_commit(b, tp_B);
            Transaction nt(t);
            nt.begin()
            try
               b._try_delete();
               nt.commit();
            handle IntegrityError
               nt.abort();
            end handle
   end case
```

Figure 7: Method delete() in interface Object.

The algorithm for delete() uses these functions:

Type(*o*) – the type, or class, of object *o*, which is the most specific type of *o* in any type hierarchy.

LbM(*tp*) – the lower bound multiplicity for *tp* in the <*association*> for the relationship represented by traversal path *tp*.

ImpB(*tp*) – the implicit destructibility binding for *tp* (minus any | symbol) in the <*association*> for the relationship represented by traversal path *tp*.

Inverse(*tp*) – the inverse traversal path of *tp*.

Refs(*o.tp*) – the number of references in *o.tp*, which, if *tp* is a collection, is the cardinality of the collection, i.e., *o.tp*.cardinality() and, if *tp* is a reference, is 0 if nil and 1 if not.

The delete() method provides a nested transaction that embeds the complex object operation, permitting its effects on the database to be undone if an exception occurs.

The _try_delete() method is an indirectly recursive method that may result in the implicit deletion of many objects that are related directly or indirectly to the object upon which it is invoked, designated here as *a*. Its invocation on an object must be dynamically bound to the method on the class representing the object's most specific type. This ensures that _try_delete() processes all traversal path instances involving the object.

The method first checks that object *a* has not already been marked for deletion by invoking the _deleted() method on the current transaction. If it has, _try_delete() simply exits. If not, it marks object *a* for deletion by invoking _mark_for_deletion().

The outer **for each** loop traverses every traversal path tp_A defined in (or inherited by) class *A*. For each such path in object *a*, the inner **for each** traverses all references in the traversal path. The purpose here is to attempt to implicitly drop each reference to a target object *b* (including the inverse reference to *a*) so that object *a* can be deleted. The code first drops each such reference by invoking the _primitive_drop_tp_A method on *a*, which drops *a*.tp_A's reference to *b* and *b*.tp_B's reference to *a*. It then invokes the method _enforce_binding() on the target object *b* to enforce the implicit destructibility binding ImpB(tp_B) for the inverse traversal path tp_B.

The last step of _try_delete() actually deletes the object but only if none of the _enforce_binding() invocations raise an exception.

The _enforce_binding() method is assumed for simplicity to be defined in the interface Object. The method for one class in a relationship must be accessible to the other class. The method enforces the destructibility binding semantics specified in Table 1. Here, *b* denotes the implicit parameter and tp_B denotes the explicit parameter since _enforce_binding() is invoked on a target object to enforce the binding for the inverse traversal path in that target object. It is invoked after a reference to target object *b* and its inverse reference in the traversal path tp_B have been dropped by the caller. The **case** statement executes the appropriate code for the given *binding*. The method _check_path_at_commit() is invoked to ensure that a lower bound constraint is rechecked at the end of the complex object operation, i.e., within commit() of the current, nested transaction.

5 CONCLUSION

In this paper, we have proposed adding ORN to the ODMG Object Model and have referenced, illustrated, and discussed algorithms for implementing ORN semantics in an ODMS. The shortcomings of our proposal are that the Object Model is made slightly more complex and ODMS implementations must include a nested transaction capability. Despite these shortcomings and regardless of whether or not ORN is added to the ODMG standard, we believe that vendors should strongly consider including ORN as an extended feature to their ODMSs. We conclude by summarizing the reasons:

- ORN is a simple notation that allows the database developer to specify a variety of association semantics, which define the scopes of complex and composite objects.
- The extended ODL would facilitate a straightforward mapping of association semantics from a conceptual database model, expressed as an ORN-extended UML class diagram, to the logical database model, expressed in the ODL.
- The ODMS would provide the same support for associations that is provided by relational DBMSs via the SQL references clause plus support even more powerful association semantics.
- If no *<association>* is given for a traversal path, the default *<association>* corresponds to current system capabilities. Thus, adding ORN is a pure extension requiring no changes to the underlying Object Model capabilities.
- The implementation of this extension is relatively simple as shown by the algorithms we have made available and their implementation in OR+.
- The benefits are increased database development productivity and improved database integrity as much less code needs to be developed and maintained by database application developers.

ACKNOWLEDGEMENTS

This work was partially supported by the NSF cooperative agreement HRD-9707076.

REFERENCES

Balaban, M. and Shoval, P., 2002. MEER – A EER model enhanced with structure methods. *Information Systems*, 27 (4), 245-275.

Bouzeghoub, M. and Metais, E., 1991. Semantic modeling and object oriented databases. In *Proc. 17th Int'l VLDB Conference*, Barcelona, Spain. 3-14.

Cattel, R.G.G., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O, Stanienda, T., and Velez, F., 2000. *The Object Database Standard: ODMG 3.0*. San Mateo, CA: Morgan Kaufmann.

Chen, P.P., 1976. The entity-relationship model: towards a unified view of data. *ACM Transactions on Database Systems*, 1(1), 1-36.

ANSI, 2003. *Information technology - Database languages - SQL, Parts 1-4*, New York, NY: American National Standards Institute (ANSI). Available from: www.ansi.org.

Ehlmann, B.K. and Riccardi, G.A., 1996. A comparison of ORN to other declarative schemes for specifying relationship semantics. *Information and Software Technology*, 38 (7), 455-465.

Ehlmann, B.K. and Riccardi, G.A., 1997. Object Relater *Plus*: A Practical Tool for Developing Enhanced Object Databases. In *Proc. 13th Int'l Conference on Data Engineering*, Birmingham, England. 412-421.

Ehlmann, B.K., Rishe, N., and Shi, J., 2000. The formal specification of ORN semantics. *Information and Software Technology*, 42 (3), 159-170.

Ehlmann, B.K., Riccardi, G.A., Rishe, N., and Shi, J., 2002. Specifying and enforcing association semantics via ORN in the presence of association cycles, *IEEE Transactions on Knowledge and Data Engineering*, 14 (6), 1249-1257.

Ehlmann, B.K. and Yu, X., 2002. Extending UML class diagrams to capture additional association semantics. In *Proc. 20th IASTED Int'l Conf. on Applied Informatics*, Innsbruck, Austria. 395-401.

Ehlmann, B.K., 2002. A data modeling tool where associations come alive. In *Proc. 21st IASTED Int'l Conf. on Modelling, Identification, and Control*, Innsbruck, Austria. 66-72. Available at www.siue.edu/~behlman.

Ehlmann, B.K., 2006. Algorithms for the implementation of ORN in an ODMG-compliant ODMS. Available from: www.siue.edu/~behlman.

Mellor, S.J., Clark, A.N., and Futagami, T., 2003. Guest editor's introduction: Model-Driven Development. *IEEE Software*, 20 (5), 19-25.

Lazarevic, B., Misic, V., 1991. Extending the entity-relationship model to capture dynamic behavior. *European Journal of Information Systems*, 1 (2), 95-106.

Progress Software, 2006. *ObjectStore Interprise*. Bedford, MA: Progress Software. Available from: www.objectstore.com/datasheet/index.ssp.

OMG, 2005. *Unified Modeling Language (UML) Specification*. Version 2.0. Object Management Group (OMG). Available from: www.uml.org.