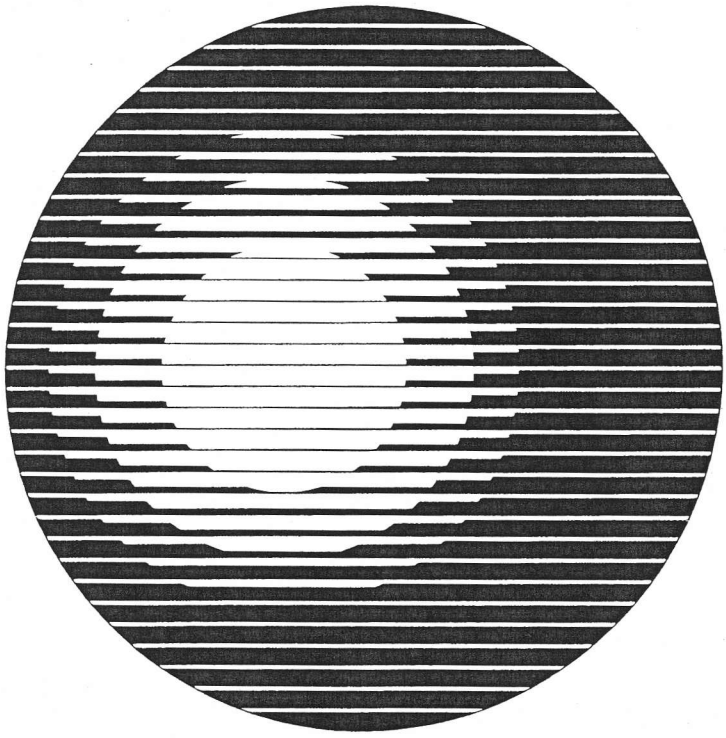
VOLUME 42 NUMBER 3 2000

# INFORMATION AND SOFTWARE TECHNOLOGY



ELSEVIER

Volume 42, Number 3, 25 February 2000

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

# Contents





0950-5849(200001)42:2;1-R

05281

ELSEVIER

Information and Software Technology 42 (2000) 159–170

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

# The formal specification of ORN semantics

B.K. Ehlmann[a,*], N. Rishe[b], J. Shi[b]

[a]*Department of Computer Information Sciences, Florida A&M University, Tallahassee, FL 32307, USA*
[b]*High-Performance Database Research Center, Florida International University, University Park, Miami, FL 33199, USA*

Accepted 7 April 1999

**Abstract**

Object Relationship Notation (ORN) is a declarative scheme that permits a variety of common types of relationships to be conveniently defined to a Database Management System (DBMS), thereby allowing the DBMS to automatically enforce their semantics. Though first proposed for object DBMSs, ORN is applicable to any data model that represents binary entity-relationships or to any DBMS that implements them. In this paper, we first describe ORN semantics informally as has been done in previous papers. We then provide a formal specification of these semantics using the Z-notation. Specifying ORN semantics via formal methods gives ORN a solid mathematical foundation. The semantics are defined in the context of an abstract database of sets and relations in a recursive manner that is precise, unambiguous, and noncircular. 

*Keywords*: Object relationship notation; Data modeling; Formal methods

## 1. Introduction

Object Relationship Notation (ORN) is a declarative scheme for defining a variety of common aggregation, or non-inheritance, relationship types—i.e. the "is part of," "is defined by," "is owned by," and "is associated with" types of relationships and their many variations. These relationships define the *class–composition hierarchy* in an object database [1]. ORN allows their semantics to be distinguished and documented during system analysis and design at an entity (or object)-relationship level of abstraction and then defined to a DBMS during implementation. This allows the early detection of relationship subtleties and inconsistencies and the automatic maintenance of relationship semantics by the DBMS, thereby improving database integrity. Significantly, this is achieved without DBMS user having to develop any programming code or any complex constraint and trigger specifications [2].

Previous papers have explored various aspects of ORN. In Ref. [3] ORN was compared to other declarative schemes for specifying relationship semantics—e.g. the REFERENCES clause of SQL, which is given for foreign keys in relational databases. This paper showed that the most unique aspects of ORN are that it provides for the enforcement of upper and lower bound cardinality constraints and, more importantly, allows delete propagation to be based on these constraints. This results in a simpler and more powerful scheme for specifying a greater variety of relationship types. In Ref. [4] an integrated methodology was presented for developing relationships in a database based on ORN. This paper showed how ORN, unlike the declarative scheme of SQL, can be incorporated into ER-like Diagrams [5]. Ehlmann and Riccardi [6] discussed ORN's implementation in an extensible, ODMG-93 compatible [7], Object DBMS prototype called Object Relater *Plus* (OR+). Hardeman [8] showed that with ORN, subtleties and inconsistencies in relationship behavior can be identified and automatically detected during analysis and design. Brown [9] presented the user interface, architecture, and features of the ORN Simulator, a database design tool. Finally, in Ref. [2] the syntax, semantics, and pragmatics for incorporating ORN into SQL were described as well as the benefits.

The syntax of ORN can be easily specified by simple syntax diagrams; however, the complete semantics of ORN are not as easily specified, especially when *relationship cycles* are considered. A relationship cycle occurs in a database when an object is related to itself, directly or indirectly. The problems posed by such cycles in specifying ORN semantics are inherent in any scheme that defines relationship semantics recursively, as does ORN. This paper describes the semantics of ORN via formal methods

* Corresponding author. Tel.: +1-850-599-3022; fax: +1-850-599-3221.
*E-mail address:* ehlmann@cis.famu.edu (B.K. Ehlmann).

0950-5849/00/$ - see front matter 
PII: S0950-5849(99)00051-8

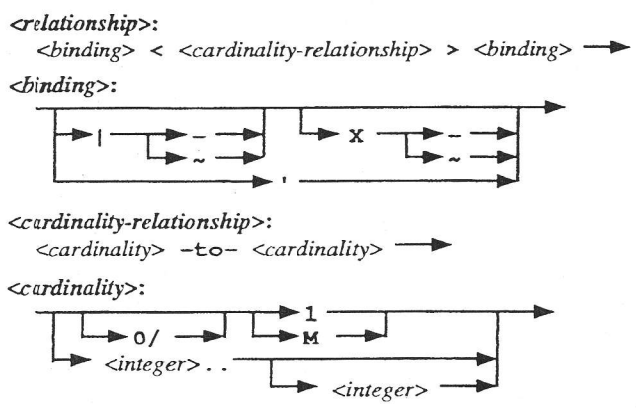



Fig. 1. ORN syntax.

in a manner that is precise, unambiguous, noncircular, and almost complete—*almost* complete because relationship cycles necessitate that a small restriction be placed on ORN to facilitate processing order independent semantics and a less complex formal specification.

The remainder of the paper is organized as follows. In Section 2 we give the syntax of ORN and an informal, English description of the semantics, similar to that which has appeared in previous papers. Section 3 makes this description mathematically precise by specifying ORN semantics using formal methods. The notation we employ is the *Z* specifications language [10]. Explanations that accompany the formal specifications should allow the reader unfamiliar with *Z* but well versed in set theory and first-order predicate calculus to understand these specifications as well as gain insight into *Z*. Section 4 briefly discusses relationship cycles—the problems they pose to our formal specification and how these problems are addressed. We conclude in Section 5 with some summary remarks.

## 2. An informal description

The syntax and semantics of ORN define a taxonomy of binary, aggregate relationship types common to databases. Fig. 1 gives the syntax of ORN. Table 1 gives the meanings for all ORN symbols.

Relationships in ORN are described on two levels. A ⟨*cardinality-relationship*⟩ defines a binary relationship type solely by class *cardinalities*. *Bindings* are then added in a ⟨*relationship*⟩ to indicate the level of binding between related objects. The level of binding determines the implicit and explicit destructibility of relationship instances and whether relationship destruction can result in the implicit deletion of related objects. Implicit destructibility of relationships is important since all existing relationships involving an object must be implicitly destroyed, or cut, before an object can be deleted. Implicit deletions of related objects, which may result from relationship destructions, enforce cardinalities and define the extent of *complex* and *composite* *objects*, objects that are closely connected with or contain other objects, respectively.

In a ⟨*cardinality-relationship*⟩, the ⟨*cardinality*⟩ before the -to- describes the cardinality for the *subject* class; the

Table 1
Meaning of ORN symbols

< > - Angle brackets distinguishing a <*relationship*> from a <*cardinality-relationship*>.

**Cardinality Symbols:**

0 - "zero" / - "or" 1 - "one"

M - "Many" meaning one or more

.. - "or more" as in 2.., meaning two or more, or "to" as in 1..9, meaning 1 to 9 inclusive

**Binding Symbols:**

Binding symbols are described in terms of an object class *C* in a relationship *R* having the given binding. Deletion of a *C* object succeeds only if all existing relationships involving that object, i.e., relationship instances, are implicitly destructible, i.e., can be *cut*. In addition, the deletion of a *C* object or explicit destruction of an *R* relationship succeeds only if all required implicit deletions succeed. "none" indicates the absence of an applicable binding symbol.

none - Default implicit destructibility binding. On delete of a *C* object, an existing *R* relationship, i.e., relationship instance, is implicitly destructible provided implicit destruction does not violate the cardinality of *C*.*

|- - Minus implicit destructibility binding. On delete of a *C* object, an existing *R* relationship is never implicitly destructible. *Implicit* relationship destruction is denoted by the |, symbolizing a *cut* in the relationship.

|~ - Propagate implicit destructibility binding. On delete of a *C* object, an existing *R* relationship is always implicitly destructible. The related object is implicitly deleted when implicit destruction violates the cardinality of *C*.

none - Default explicit destructibility binding. An *R* relationship, i.e., relationship instance, is explicitly destructible provided explicit destruction does not violate the cardinality of *C*.*

X- - Minus explicit destructibility binding. An *R* relationship is never explicitly destructible. *Explicit* relationship destruction is denoted by the X.

X~ - Propagate explicit destructibility binding. An *R* relationship is always explicitly destructible. The related object is implicitly deleted when explicit destruction violates the cardinality of *C*.

' - Prime implicit and explicit destructibility binding. On delete of a *C* object, an existing *R* relationship is implicitly destructible. An implicit delete is done on the related, i.e., subordinate, object. An *R* relationship is always explicitly destructible. Again, an implicit delete is done on the subordinate object. The implicit deletion of a subordinate object is required, and thus must succeed, if and only if relationshp destruction, implicit or explicit, violates the cardinality of *C*.*

* The check for a violation caused by the destruction is deferred until the end of the current operation.

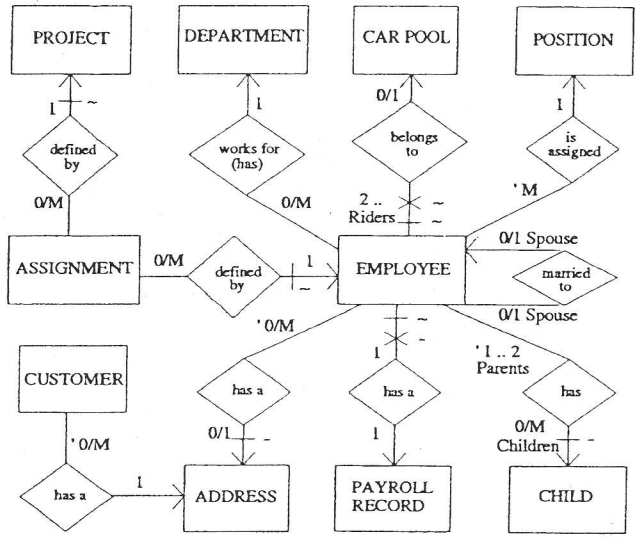



Fig. 2. ORD for a company database.

⟨*cardinality*⟩ after the -to- describes the cardinality for the *related* class. The subject class cardinality is the number of objects of the subject class that can relate to a single object of the related class. Likewise, the related class cardinality is the number of objects of the related class that can relate to a single object of the subject class. For example, the relationship between departments and employees could be described as 1-to-0/M. Each object of type department, the subject class, relates to zero or to many (meaning one or more) employees. Each object of type employee, the related class, relates to exactly one department.

In a ⟨*relationship*⟩, the ⟨*binding*⟩ before the ⟨ indicates the binding for the subject class; the one after the ⟩ indicates the binding for the related class. Relationship semantics are derived from the cardinality semantics and the semantics of the given bindings.

Fig. 2 shows how ORN is incorporated into an Object–Relationship Diagram (ORD), which is similar to an ERD. Fig. 3 shows how it is incorporated into the Object Database Definition Language (ODDL), developed as part of OR+ [6]. Fig. 3 gives partial ODDL specifications corresponding to the ORD in Fig. 2. These specifications refine the objects and relationships described in an ORD by defining *literal-valued attributes*, *object-valued attributes*, and methods for each object class—e.g. SSN, Dept, and RaiseSalary, respectively, for the class employee. The Object Database Manipulation Language (ODML) of OR+ provides for database creation, access, and manipulation based on ODDL.

To illustrate the use of ORN to describe relationship semantics, we will discuss some variations on the classic departments–employees relationship. Although the semantics of some of these variations may seem odd for this relationship, they make sense for other one-to-many relationships, some of which appear in Fig. 2. The discussion below while focusing on a one-to-many relationship is also relevant to one-to-one and many-to-many relationships as

```
Database CompanyDB  // Company DataBase
{
  class employee {
    d_String      SSN;  // Soc. Sec. No.
    d_String      Name; // Last, First name
    Date          BirthDate;
    department    Dept inverse Employees <0/M-to-1>;
    Set<assignment> Assignments inverse Employees
                        |~<1-to-0/M>;
    employee      Spouse inverse Spouse <0/1-to-0/1>;
    List<child>   Children inverse Parents '<1..2-to-0/M>|-;
    car_pool      CarPool inverse Riders |~X~<2..-to-0/1>;
    ...
$C++
    void RaiseSalary(int percentage);
    void CancelAssignments();
    ...
$.
  };
  extent Set<employee> Employees  key SSN;

  class department {
    d_String          Name;
    d_String          Location;  // Building No.
    Set<employee>     Employees inverse Dept;
                        // <1-to-0/M> by default
    ...
  };
  extent Set<department> Departments key Name;

  class car_pool {
    d_String          LicNum;  // License Number
    Set<employee>     Riders inverse CarPool;
    ...
  };
  extent Set<car_pool> CarPools key (LicNum);
...
};
```

Fig. 3. Partial ODDL specifications for ORD in Fig. 2.

cardinality and binding semantics for one side of a relationship are mostly independent of those for the other side.

1. ⟨1-*to*-0/*M*⟩. This is the relationship between departments and employees as shown in Fig. 2. The default implicit destructibility binding for the employee class, i.e. the absence of a binding symbol after the ⟩, denotes that the relationship between an employee and a department can be (and is) implicitly destroyed or cut when an employee is deleted. Such destruction never violates the 0/M cardinality for the employee class. The default implicit destructibility binding for the department class denotes that a department cannot be deleted if it has any employees since implicit relationship destruction would violate the 1 cardinality for the department class—i.e. an employee must have 1 department. A department can be deleted only if all of its employees were deleted or reassigned to other departments.

   Finally, the default explicit destructibility binding for the employee class means that a relationship between an

employee and department can be explicitly destroyed since this would never violate the 0/M cardinality; however, the default explicit destructibility binding for the department class prohibits this destruction as it would violate the 1 cardinality. An employee's department may, however, be changed as this would not violate the 1 cardinality.

2. ⟨0/1-*to*-0/*M*⟩|-. An employee can now exist without being in a department. An employee who is in a department cannot be deleted as this would require implicit destruction, or a cut, of the relationship, which is disallowed by the |- implicit destructibility binding. An employee's relationship with a department would have to be explicitly destroyed, which is now allowed by the 0/1 cardinality, before the employee could be deleted.
3. ⟨1-*to*-*M*⟩| ~ . Every department must have at least one employee, and the | ~ implicit destructibility binding means that deletion of the last employee in a department would cause the deletion of the department. If the relationship were | ~ ⟨1-to-M⟩| ~ , then deletion of a department would cause deletion of all employees in the department. If ⟨0/1-to-3..⟩| ~ , then deletion of the third last employee would cause deletion of the department. If | ~ ⟨1-to-3..⟩| ~ , deletion of the third last employee would cause deletion of the department and its two remaining employees as well!
4. ⟨0/1-*to*-0/*M*⟩*X*-. A relationship instance between an employee and a department once created cannot be explicitly destroyed. It can, however, be implicitly destroyed if the employee is deleted. The X- binding when given applies to both classes in a ⟨*relationship*⟩.
5. (0/1-*to*-*M*⟩*X* ~ . Now, explicit destruction of the relationship between a department and the last employee in the department would cause the deletion of the department. If the relationship were X ~ ⟨1-to-M⟩X ~ , then explicit destruction of a department and employee relationship would cause deletion of the employee. And of course, if the employee is the last one in the department, then the department would also be deleted.
6. ′⟨1-*to*-0/*M*⟩. Here department is the prime class and employee the subordinate. When a department is deleted, all of the relationships it has with its employees are implicitly destroyed and an implicit delete is done on all of these employees. If any of these deletes fail, the department delete fails since the 1 cardinality constraint must be maintained. (If the relationship between departments and employees were ′⟨0/M-to-M⟩, then failure of an employee delete would not cause failure of the department delete. The employee may be the lone subordinate object of another department and, therefore, cannot yet be deleted.) Also, when the relationship between a department and an employee is explicitly destroyed, an implicit delete is done on the employee with analogous failure semantics.

Listed below are some of the relationship semantics defined in Fig. 2 by ORN. Relationship semantics associated with an employee object make it a very *complex object*.

- If an employee is deleted, all assignments for that employee are deleted as is his/her payroll record.
- If an employee is deleted, the employee's address is deleted, unless it is also the address of another employee or customer. The employee's position is also deleted, unless it is also held by another employee, and all children of the employee are deleted unless the employee's spouse also works for the company. These semantics result from the ′ (prime) bindings.
- If an employee is deleted who is one of only two riders in a car pool, the car pool is deleted. This also occurs if this employee is not deleted but his relationship with the car pool is destroyed. That is, a car pool "is defined by" two or more riders.
- An employee's relationship to a payroll record can never be explicitly destroyed.

## 3. A formal specification

In this section we utilize formal methods, specifically the formal mathematical notation of *Z*, to more fully and precisely describe ORN semantics.

The formal specification of ORN accounts for the possibility of relationship cycles, which were not mentioned in the previous section. A relationship cycle exists among objects $x_1, x_2, \ldots, x_n, n \geq 1$, when $x_1 \mapsto x_2, x_2 \mapsto x_3, \ldots, x_{n-1} \mapsto x_n$, and $x_n \mapsto x_1$, where $\mapsto$ means that the left object is related to the right object via some relationship. Such cycles, when present, result in circularities and ambiguities in the English description of ORN semantics.

The ambiguities will spill over into the formal specifications unless a small restriction is placed on ORN. The restriction is that a single |- binding cannot be given for any relationship involved in a relationship cycle, i.e. a |- binding must be given for both sides—subject and related class—in such a relationship or not at all. We impose this restriction in the formal specification by constraining the database from having a *cyclic, one-sided* |- *binding*. We discuss this restriction further in Section 4.

We also make some assumptions to simplify the formal specification. First, we assume a database having a consistent metadatabase wherein the types of objects and relationships are predefined and unchanging during the course of database operations, i.e. schema evolution and data reorganization are of no concern. Second, we assume that certain capabilities available in OR+ [6] are restricted. There are no "is a" relationships, i.e. relationships are not inherited from super or base objects; no objects are marked as "not explicitly deletable;" and there is no RXCmode (Relationship eXChange mode), a mode that suspends lower bound cardinality checks in order to do relationship exchanges. These restricted capabilities do not affect the basic semantics

of ORN. They just make their formal specification more complex.

Also, to simplify the formal specification, we assume the database operations that define ORN semantics—object creation and deletion and relationship creation, destruction, and change—operate under a simplified transaction model. This we must formally define since ORN semantics require a transaction commit operation to provide deferred checking of lower bound cardinality constraints for relationships. Although not explicitly enforced by the formal specifications, the reader should assume that a commit or abort operation, as defined, is done immediately after an object delete or a relationship destroy or change operation. This protocol ensures that these complex object operations are either completely successful or rolled back before another database operation is begun. The reader should also assume that the database can never be "closed" in transaction state, although no database close (or open) operation is formally defined. This ensures that with our simplified transaction model a final commit or abort will eventually be done, resulting in a consistent database.

In OR+, ORN is implemented within a more complex transaction model in which a complex object operation is done in an implementation-supplied transaction nested in a user-supplied transaction. If unsuccessful, the operation is automatically rolled back and an exception results.

Finally, again for brevity, we will not provide operation schema to handle error situations. We will simply assume that violations to preconditions or constraint predicates on the database result in appropriate exceptions.

To specify ORN semantics via Z, we must first establish the context in which these semantics operate. That is, we must formally define a database. The database is formally defined in terms of sets of classes, objects, relationships, and instances, where relationships and instances can be directly mapped to relations and ordered pairs.

### 3.1. Defining a database

*Class* is the given set of all object classes. *Object* is the set of all objects.

*[Class]*
*Object* ≙ *[type: Class]*

The *type* of each object associates it with a particular class. In the horizontal schema defined for *Object* (denoted by the ≙ and the []s), *type* is declared a variable of type *Class*, since the values of *type* are members of the set *Class*. Both the *Class* and *Object* sets can be used as types.

A *cardinal* and a *binding*, or more precisely the members of these sets, are used to record cardinalities and bindings, respectively, in an *ORN*. An *ORN* is an encoded ⟨*relationship*⟩.

$cardinal = \mathbb{N} \cup \{M\}$
*binding* ::= default | ~ | - | '

*ORN*

$sLB, sUB, rLB, rUB$: *cardinal*
*sImpB, sExpB, rImpB, rExpB*: *binding*

---

$(sImpB = ' \Leftrightarrow sExpB = ') \wedge (rImpB = ' \Leftrightarrow rExpB = ')$
$\wedge (sExpB = - \Leftrightarrow rExpB = -)$
$sLB \neq M \wedge rLB \neq M \wedge (sUB = M \vee sLB \leq sUB)$
$\wedge (rUB = M \vee rLB \leq rUB)$

The set *cardinal* includes natural numbers (denoted by $\mathbb{N}$) and the letter M, meaning "many." ( $\cup$ denotes set union.) The data type *binding* is a set containing four enumerated values: default, ~, -, and '.

A vertical schema, like that given above for *ORN*, declares variables in the top part and may express predicates in the bottom part that constraint the values of these variables. Within the variable names in the *ORN* schema, *s* refers to the subject class, *r* to the related class, *LB* to lower bound, *UB* to upper bound, *ImpB* to implicit binding, and *ExpB* to explicit binding. The two predicates given in the *ORN* schema express constraints on the bindings and cardinalities given in a ⟨*relationship*⟩. For example, the subject class implicit binding is ' if and only if (denoted by $\Leftrightarrow$) the subject class explicit binding is '. ( $\wedge$ denotes "logical and" and $\vee$ "logical or.")

*Relationship* is the set of all binary relationships between a subject class (*sC*) and a related class (*rC*). Each relationship is described by an *ORN*.

*Relationship* ≙ *[sC,rC: Class; desc: ORN]*

*Instance* is the set of all instances of binary relationships between two objects, one object designated as the subject object (*sO*) and the other as the related object (*rO*). The *type* of an *Instance* associates it with a specific *Relationship*.

*Instance* ≙ *[sO,rO: Object; type: Relationship|sO.type = type.sC ∧ rO.type = type.rC]*

The predicate part above (which follows a | in a horizontal schema), states that the type of the subject and related objects in an instance must be that of the subject and related classes of the associated relationship, respectively.

The functions *relation*, *relation_type*, and *ordered_pair* are not used in subsequent specifications but are defined below to show how relationships and instances are mathematically related to relations, relation types, and ordered pairs, respectively.

$$relation: Relationship \rightarrow (Object \leftrightarrow Object)$$

$$\forall r: Relationship \bullet relation(r) = \{i: Instances \mid i.type = r \bullet i.sO \mapsto i.rO\}$$

$$relation\_type: Relationship \rightarrow \mathbb{P}\,(Object \leftrightarrow Object)$$

$$\forall r: Relationship \bullet relation\_type(r) = \mathbb{P}\,\{o1, o2: Object \mid o1.type = r.sC \wedge o2.type = r.rC \bullet o1 \mapsto o2\}$$

$$ordered\_pair: Instance \rightarrow Object \times Object$$

$$\forall i: Instance \bullet ordered\_pair(i) = sO \mapsto rO$$

The functions above are specified by axiomatic descriptions. In the first such description above, the top part declares *relation* to be the name of a total function (denoted by $\rightarrow$) that takes a *Relationship* for its argument and has a relation as its value. ($X \leftrightarrow Y$ denotes a relation between sets $X$ and $Y$.) The bottom part of the axiomatic description is a predicate that fixes the value of *relation* for any argument. (The notation $\forall D \bullet P$ asserts that for all values of the variables declared in $D$, predicate $P$ is true. $\{D \mid P \bullet E\}$ denotes a set consisting of all values of term $E$ for all values of the variables declared in $D$ constrained by predicate $P$. $x \mapsto y$ denotes an ordered pair $(x, y)$.). The function *relation_type* maps a *Relationship* to a relation type. ($\mathbb{P}S$ denotes the power set of set $S$.) The function *ordered_pair* maps an *Instance* to an ordered pair. ($X \times Y$ denotes the Cartesian product of sets $X$ and $Y$.)

The inverse functions below return the inverses of a given *ORN*, *Relationship*, and *Instance*, respectively. The inverse operator is $\sim$. (_ denotes the given operand or argument.)

$$\_^{\sim}: ORN \rightarrow ORN$$

$$\forall x, y: ORN \bullet (y = x^{\sim}) \Leftrightarrow (y.sLB = x.rLB \wedge y.sUB = x.rUB \wedge y.rLB = x.sLB \wedge y.rUB = x.sUB \wedge y.sImpB = x.rImpB \wedge y.sExpB = x.rExpB \wedge y.rImpB = x.sImpB \wedge y.rExpB = x.sExpB)$$

$$\_^{\sim}: Relationship \rightarrow Relationship$$

$$\forall x, y: Relationship \bullet (y = x^{\sim}) \Leftrightarrow (y.sC = x.rC \wedge y.rC = x.sC \wedge y.desc = (x.desc)^{\sim})$$

$$\_^{\sim}: Instance \rightarrow Instance$$

$$\forall x, y: Instance \bullet (y = x^{\sim}) \Leftrightarrow (y.sO = x.rO \wedge y.rO = x.sO \wedge y.type = (x.type)^{\sim})$$

The recursive function *related* returns *true* or *false* for two given objects and a set of instances to indicate whether the two objects are related either directly or indirectly via the set of instances.

$$related: Object \times Object \times \mathbb{P}\,Instance \rightarrow \{true, false\}$$

$$\forall o1, o2: Object;\ i\_set: \mathbb{P}\,Instance \bullet related(o1, o2, i\_set) = (\exists i1: Instance \bullet i1 \in i\_set \wedge i1.sO = o1 \wedge i1.rO = o2) \vee (\exists i2: i\_set;\ o: Object \bullet i2 \in i\_set \wedge i2.sO = o1 \wedge i2.rO = o \wedge related(o, o2))$$

The related function is used later to detect a relationship cycle in the database. (The notation $\exists D \bullet P$ asserts that there exists values for variables declared in $D$ such that predicate $P$ is true.)

A metadatabase, *MetaDB*, is a set of classes and relationships.

*MetaDB*

$$classes: \mathbb{P}\,Class$$
$$relationships: \mathbb{P}\,Relationship$$

$$\forall r: relationships \bullet r.sC \in classes \wedge r.rC \in classes$$
$$\forall r1: relationships \bullet (\exists r2: relationships \bullet r1 = r2^{\sim})$$

All relationships in the metadatabase are defined over the classes in the metadatabase. All relationships in the metadatabase have an inverse relationship also in the metadatabase.

A database is assumed to have a consistent metadatabase, which is declared as a global variable *mdb*.

$$mdb: MetaDB$$

A database, or *DB*, consists of a set of objects and a set of relationship instances. The state of this abstract data type is given by the following schema.

*DB*

$$objects: \mathbb{P}\,Object$$
$$instances: \mathbb{P}\,Instance$$

$$(\exists n: \mathbb{N} \bullet \#objects < n) \wedge (\exists m: \mathbb{N} \bullet \#instances < m)$$
$$(\forall o: objects \bullet o.type \in mdb.classes) \wedge (\forall i: instances \bullet i.type \in mdb.relationships)$$
$$\forall i: instances \bullet i.sO \in objects \wedge i.rO \in objects$$
$$\forall i1, i2: instances \bullet (i1.type = i2.type \wedge i1 \neq i2) \Rightarrow (i1.sO \neq i2.sO \vee i1.rO \neq i2.rO)$$
$$\forall i1: instances \bullet (\exists i2: instances \bullet i1 = i2^{\sim})$$
$$\forall o: objects \bullet (\forall r: mdb.relationships \mid r.sC = o.type \bullet (r.rUB = \mathrm{M} \vee \#\{i: db.instances \mid i.type = r \wedge i.sO = o\} \leq r.desc.rUB)$$
$$\neg\exists r: mdb.relationships \mid r.desc.sImpB = - \wedge r.desc.rImpB \neq - \bullet \exists i: db.instances \mid i.type = r \bullet i.sO = i.rO \vee related(i.sO, i.rO, db.instances \setminus \{i, i^{\sim}\})$$

The number of objects and instances in the database is

finite. (#$S$ denotes the cardinality of set $S$.) All objects in a database belong to (or have type of) one of the classes in the metadatabase, and all instances are instances of one of the relationships in the metadatabase. The instances in the database are between the objects in the database. All instances of a particular relationship are unique. ($\Rightarrow$ denotes implication.) Each instance in the database has an inverse instance in the database. All upper bound relationship cardinality constraints are satisfied. (The notation $\forall D|P \bullet Q$ asserts that for all values of the variables declared in $D$ constrained by predicate $P$, the predicate $Q$ is true. $\forall D|P \bullet Q$ is equivalent to $\forall D \bullet (P \Rightarrow Q)$.) Finally, there are no cyclic, one-sided |- bindings. That is, for any instance of a relationship having a one-sided |- binding, the subject and related objects are not identical or are not related via other instances. (The notation $\exists D|P \bullet Q$ asserts that there exists values for variables declared in $D$ constrained by predicate $P$ such that predicate $Q$ is true. $\exists D|P \bullet Q$ is equivalent to $\exists D \bullet (P \wedge Q)$.)

An *InitDB* operation specifies the initial state of the database. The schema for this abstract operation is given below in horizontal format.

$$InitDB \triangleq [DB'|objects' = \varnothing \wedge instances' = \varnothing]$$

The declaration $DB'$ includes into the operation schema the declarations and predicates of the *DB* schema. The $'$ indicates that only the ending state of the database, i.e. the *DB* component values after the operation completes, will be specified. At the end of this operation, both the *objects* and *instances* sets of *DB* are empty.

### 3.2. Defining a transaction model

In the definition for *DB* no predicate is given to ensure that all lower bound relationship cardinality constraints are satisfied. This is because these constraints are only enforced when a transaction is committed. We now define a transaction model that provides a transaction commit operation.

*TransData* consists of a flag, indicating whether or not "the system is in transaction state", and a saved copy of the database as it was at the beginning of a transaction. The *InitTransData* operation specifies the initial state of *TransData*.

$$TransData \triangleq [inTrans: \{true,false\};\ savedDB: DB]$$
$$InitTransData \triangleq [Transaction'|inTrans' = false]$$

The operation schemas for transaction operations are given below in vertical format followed by some explanatory remarks.

*BeginTransaction*

$\Delta TransData$
$\Xi DB$

---

$inTrans = false$
$savedDB' = \theta DB$
$inTrans' = true$

In beginning a transaction, *TransData* will change, thus the declaration $\Delta TransData$. This includes into the operation schema two versions of the *TransData* schema: the first having undashed variables, representing the starting state of *TransData*, i.e. component values prior to the operation, and the second having dashed ($'$ ed) variables, representing the ending state of *TransData*, i.e. component values after the operation completes. The declaration $\Xi DB$ includes the *DB* schema into the *BeginTransaction* operation schema, and the $\Xi$ symbol specifies that the state of *DB* is not changed by the operation.

The first predicate in the *BeginTransaction* schema states that the system is not in transaction state. If this precondition is not met, we assume an exception results. This invalid case for beginning a transaction is not formally specified.

The first of two postconditions specifies that the copy of the current database is saved. That is, after this operation is completed, the ending value of *savedDB* in *TransData*, denoted by $savedDB'$, is equal to the value of the current database. ($\theta DB$ is a term of type *DB* denoting the database in its starting state, which for this operation is the same as the database in its ending state, $\theta DB'$.) The second postcondition indicates that the system is now in transaction state.

*CommitTransaction*

$\Delta Transaction$
$\Xi DB$

---

$inTrans$
$\forall o: objects \bullet (\forall r: mdb.relationships \mid r.sC = o.type \bullet$
$\quad \#\{i: instances \mid i.type = r \wedge i.sO = o\} \geq r.desc.rLB)$
$inTrans' = false$

A *CommitTransaction* requires the database to be in transaction state. All lower bound relationship cardinality constraints must be satisfied. If both preconditions are true, the database is no longer in transaction state, which is specified by the last predicate.

*AbortTransaction*

$\Delta Transaction$
$\Delta DB$

---

$inTrans$
$\theta DB' = savedDB$
$inTrans' = false$

The database is in transaction state. At completion of this operation, the database is as it was at the beginning of the transaction, and it is no longer in transaction state.

### 3.3. Defining needed functions

Before formally defining the operation schemas for updating a database, a number of functions on a database must be defined, where the database is given as the first

argument. Certain of these functions are recursive and dependent on another argument, the set of objects that have already been traversed. This set can be thought of as the "deleted object set" (*dos*), the set of objects that have already been "marked for deletion." The set is used to detect a relationship cycle and terminate the recursion and is discussed further in Section 4. The functions in this subsection may be studied now in bottom–up fashion or top–down as they are invoked from the operation schemas given in the next subsection.

The function *related_instances* returns the set of all instances associated with a given set of objects.

$$related\_instances: DB \times \mathbb{P}\ Object \nrightarrow \mathbb{P}\ Instance$$

$$\forall db: DB;\ os: \mathbb{P}\ Object \mid os \subseteq db.objects \bullet related\_instances(db, os) = \{i: db.instances \mid i.sO \in os \vee i.rO \in os\}$$

This is a partial function since it is only defined for a given object set, the second argument, that is a subset of the objects in the given database, the first argument. ($\nrightarrow$ denotes a partial function.)

The function *sLB_violated* returns *true* or *false* for a given instance to indicate whether the subject class lower bound cardinality is violated if the given instance is destroyed.

$$sLB\_violated: DB \times Instance \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance \mid i \in db.instances \bullet sLB\_violated(db, i) = \#\{i2: db.instances \mid i2.type = i.type \wedge i2.rO = i.rO\} = i.type.desc.sLB$$

The function *sUB_violated* returns *true* or *false* for a given instance to indicate whether the subject class upper bound cardinality is violated if the given instance is created.

$$sUB\_violated: DB \times Instance \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance \mid i \in db.instances \bullet sUB\_violated(db, i) = \#\{i2: db.instances \mid i2.type = i.type \wedge i2.rO = i.rO\} = i.type.desc.sUB$$

The recursive function *rO_implicitly_deletable* returns *true* or *false* for a given instance to indicate whether the related object can be implicitly deleted when the given instance is destroyed.

$$rO\_implicitly\_deletable: DB \times Instance \times \mathbb{P}\ Object \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance;\ dos: \mathbb{P}\ Object \mid i \in db.instances \wedge dos \subseteq db.objects \bullet rO\_implicitly\_deletable(db, i, dos) = (i.rO \in dos \vee \forall i2: db.instances \mid i.rO \notin dos \wedge i2.sO = i.rO \wedge i2 \neq i^{\sim} \bullet implicitly\_destructible(db, i2, dos \cup \{i.rO\}))$$

The recursive function *implicitly_destructible* returns *true* or *false* for a given instance to indicate whether the given instance is implicitly destructible with respect to the subject object in the instance.

$$implicitly\_destructible: DB \times Instance \times \mathbb{P}\ Object \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance;\ dos: \mathbb{P}\ Object \mid i \in db.instances \wedge dos \subseteq db.objects \bullet implicitly\_destructible(db, i, dos) = ($$
$$(i.type.desc.sImpB = \ - \ \wedge \text{false}) \vee$$
$$(i.type.desc.sImpB = \text{default} \wedge \text{true}) \vee$$
$$(i.type.desc.sImpB = \ \sim \ \wedge (\neg sLB\_violated(db, i) \vee rO\_implicitly\_deletable(db, i, dos))) \vee$$
$$(i.type.desc.sImpB = \ ' \ \wedge (\neg sLB\_violated(db, i) \vee rO\_implicitly\_deletable(db, i, dos))))$$

The function *explicitly_destructible* returns *true* or *false* for a given instance to indicate whether the given instance is explicitly destructible from the perspective of the subject object in the instance.

$$explicitly\_destructible: DB \times Instance \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance \mid i \in db.instances \bullet explicitly\_destructible(db, i) = ($$
$$(i.type.desc.sExpB = \ - \ \wedge \text{false}) \vee$$
$$(i.type.desc.sExpB = \text{default} \wedge \text{true}) \vee$$
$$(i.type.desc.sExpB = \ \sim \ \wedge (\neg sLB\_violated(db, i) \vee rO\_implicitly\_deletable(db, i, \varnothing))) \vee$$
$$(i.type.desc.sExpB = \ ' \ \wedge (\neg sLB\_violated(db, i) \vee rO\_implicitly\_deletable(db, i, \varnothing))))$$

The function *deletable* returns *true* or *false* for a given object to indicate whether the given object can be explicitly deleted.

$$deletable: DB \times Object \nrightarrow \{true, false\}$$

$$\forall db: DB;\ o: Object \mid o \in db.objects \bullet deletable(db, o) = \forall i: db.instances \mid i.sO = o \bullet implicitly\_destructible(db, i, \{o\})$$

The recursive function *drO* returns *true* or *false* depending on whether the related object of the given instance is a *dependent related object* based on the given subject class binding. Such an object is deleted when the given instance is destroyed.

$$drO: DB \times Instance \times binding \times \mathbb{P}\ Object \nrightarrow \{true, false\}$$

$$\forall db: DB;\ i: Instance;\ b: binding;\ dos: \mathbb{P}\ Object \mid i \in db.instances \wedge dos \subseteq db.objects \bullet drO(db, i, b, dos) = ((b = \sim \wedge sLB\_violated(db, i) \wedge rO\_implicitly\_deletable(db, i, dos)) \vee (b = ' \wedge rO\_implicitly\_deletable(db, i, dos)))$$

The recursive function *complex_rO* returns the *complex object set* for the related object of the given instance. This

complex object set is the set of all objects that must be deleted when this related object is implicitly deleted on destruction of the given instance.

$$complex\_rO: DB \times Instance \times \mathbb{P}\ Object \nrightarrow \mathbb{P}\ Object$$

$$\begin{aligned}&\forall db: DB;\ i: Instance;\ dos: \mathbb{P}\ Object \mid i \in db.instances \wedge\\ &\quad dos \subseteq db.objects \bullet complex\_rO(db, i, dos) = (\{i.rO\} \cup\\ &\quad \{o: db.objects \mid \exists i1: db.instances \mid i1.rO \notin dos \bullet\\ &\qquad i1 \neq i \wedge i1 \neq i^{\sim} \wedge i1.sO = i.rO \wedge\\ &\qquad drO(db, i, i.type.desc.sImpB, dos \cup \{i.rO\}) \wedge\\ &\qquad o \in complex\_rO(db, i1, dos \cup \{i.rO\})\})\end{aligned}$$

The function *complex_drO* returns the *complex object set* for a dependent related object of the given instance. This related object is one that must be deleted if the instance is explicitly destroyed. The function returns $\varnothing$ if the related object is not a dependent object.

$$complex\_drO: DB \times Instance \nrightarrow \mathbb{P}\ Object$$

$$\begin{aligned}&\forall db: DB;\ i: Instance \mid i \in db.instances \bullet\\ &\quad complex\_drO(db, i) = \{o: db.objects \mid drO(db, i,\\ &\qquad i.type.desc.sExpB, \varnothing) \wedge o \in complex\_rO(db, i, \varnothing)\}\end{aligned}$$

The function *complex_object* returns the *complex object set* for a given object. The complex object set is the set of all objects that must be deleted when the given object is deleted.

$$complex\_object: DB \times Object \nrightarrow \mathbb{P}\ Object$$

$$\begin{aligned}&\forall db: DB;\ o: Object \mid o \in db.objects \bullet\\ &\quad complex\_object(db, o) = (\{o\} \cup \{o1: db.objects \mid\\ &\qquad \exists i: db.instances \mid i.sO = o \wedge i.rO \neq o \wedge drO(db, i,\\ &\quad i.type.desc.sImpB, \{o\}) \bullet o1 \in complex\_rO(db, i, \{o\})\})\end{aligned}$$

### 3.4. Defining the database operations

Now we are finally ready to specify the database operations that define ORN semantics. Their operation schemas, given below, along with some explanatory remarks complete the formal description of ORN semantics.

*CreateObject*

$$\begin{aligned}&\Delta DB\\ &\Xi TransData\\ &o?: Object\\ &---\\ &inTrans \wedge o? \notin objects \wedge o?.type \in mdb.classes\\ &objects' = objects \cup o?\end{aligned}$$

In creating an object the database may change, thus the declaration $\Delta DB$. *TransData* will not change, thus the $\Xi$. The object to be created, *o?* of type *Object*, is declared as input to the operation, denoted by the *?*.

The first predicate in the *CreateObject* schema, a precondition, states that the system is in transaction state, the *o?* object must not be in the database, and its type must be that of a class defined in the metadatabase. Again, if a precondition is not met, we assume an exception results. The postcondition states that the object is added to the set of objects in the database.

*CreateRelationship*

$$\begin{aligned}&\Delta DB\\ &\Xi TransData\\ &i?: Instance\\ &---\\ &inTrans \wedge i? \notin instances \wedge i?.type \in mdb.relationships\\ &i?.sO \in objects \wedge i?.rO \in objects\\ &\neg sUB\_violated(\theta DB, i?) \wedge \neg sUB\_violated(\theta DB, i?^{\sim})\\ &instances' = instances \cup \{i?, i?^{\sim}\}\end{aligned}$$

The system is in transaction state, the relationship instance to be created, i.e. added to the database, must not already be in the database, and its type must be that of a relationship defined in the metadatabase. The subject object and related object in the instance must be objects in the database. Creating the instance must not violate the upper bound cardinality constraints associated with the subject class or related class in the relationship. (Lower bound cardinality constraints are verified in an eventual *CommitTransaction* operation.) A postcondition specifies that the given instance and its inverse are added to *instances*.

*DeleteObject*

$$\begin{aligned}&\Delta DB\\ &\Xi TransData\\ &o?: Object\\ &---\\ &inTrans \wedge o? \in objects \wedge deletable(\theta DB, o?)\\ &objects' = objects \setminus complex\_object(\theta DB, o?)\\ &instances' = instances \setminus\\ &\quad related\_instances(\theta DB, complex\_object(\theta DB, o?))\end{aligned}$$

The system is in transaction state and the *o?* object must be in the database and deletable. The first of two postconditions specifies that the complex object, i.e. the set of objects that must be deleted when *o?* is deleted, is removed from the database. ($\setminus$ denotes set difference.) All related instances, i.e. those involving the objects making up the complex object, are also removed from the database. This is specified by the second postcondition.

A *CommitTransaction* done immediately after a *DeleteObject*—a protocol assumed for this as well as the *DestroyRelationship* and *ChangeRelationship* operations, given below—verifies that instances implicitly destroyed by the

complex object operation do not cause lower bound relationship cardinality violations. If the commit results in exceptions, an *AbortTransaction* must be done.

*DestroyRelationship*

$\Delta DB$
$\Xi TransData$
$i?$: *Instance*

---

$inTrans \wedge i? \in instances$
$explicitly\_destructible(\theta DB, i?) \wedge$
$\quad explicitly\_destructible(\theta DB, i?^{\sim})$
$objects' = objects \setminus (complex\_drO(\theta DB, i?) \cup$
$\quad complex\_drO(\theta DB, i?^{\sim}))$
$instances' = instances \setminus (\{i?, i?^{\sim}\} \cup$
$\quad related\_instances(\theta DB, complex\_drO(\theta DB, i?)) \cup$
$\quad related\_instances(\theta DB, complex\_drO(\theta DB, i?^{\sim}))\ )$

The system is in transaction state and the given instance to be destroyed must be in the database. It must be explicitly destructible from the perspective of both the subject and related object. The sets of objects representing any complex dependent related object and any complex dependent subject object of the given instance (or related object of its inverse) are removed from *objects*. These sets contain the objects that must be implicitly deleted when the instance is destroyed. The given instance and its inverse as well as all instances related to any implicitly deleted objects are removed from *instances*.

*ChangeRelationship*

$\Delta DB$
$\Xi Transaction$
$old?, new?$: *Instance*

---

$inTrans \wedge old? \in instances \wedge new? \notin instances \wedge$
$\quad old?.type = new?.type$
$new?.sO = old?.sO \wedge new?.rO \in objects$
$explicitly\_destructible(\theta DB, old?) \wedge$
$\quad \neg sUB\_violated(\theta DB, new?)$
$objects' = objects \setminus complex\_drO(\theta DB, old?)$
$instances' = (\ instances \setminus (\{old?, old?^{\sim}\} \cup$
$\quad related\_instances(\theta DB, complex\_drO(\theta DB, old?)))\ ) \cup$
$\qquad \{new?, new?^{\sim}\}$

The system is in transaction state, the old instance to be changed must be in the database, the new instance must not, and the type of the old and new instances must be the same. The subject object of the old and new instance must be the same, and the related object in the new instance must be an object in the database. The old instance must be explicitly destructible from the perspective of the subject object, and creating the new instance must not violate the upper bound cardinality constraints associated with the subject class. The set of objects representing any complex dependent (old) related object are removed from *objects*. This set contains the objects that must be implicitly deleted when the old instance is destroyed. The given old instance and its inverse as well as all instances related to any implicitly deleted objects are removed from *instances*, and the given new instance and its inverse are added to *instances*.

## 4. Relationship cycles

Relationships cycles in a database pose two potential problems in specifying relationship semantics. These problems have been studied by others in the context of relational databases and SQL [11,12]. One problem involves circularity and the other ambiguity. In this section, we briefly discuss these problems and how they are addressed by the formal specification of ORN semantics given in the previous section. Relationships cycles and their impact on the implementation of ORN are more fully examined in Ref. [13].

The first problem with relationship cycles is that the recursion inherent in the semantics of ORN, and often in other relationship declarative schemes, can result in circularity unless there is some means to detect a relationship cycle. This can be illustrated by the relationship cycle show in Fig. 4. Here there are just two objects, $x1$ and $y2$, within the database and two instances, $y2 \leftrightarrow x1$ of $R1$ and $y2 \leftrightarrow x1$ of $R2$ (actually four instances when inverse relationships and instances are considered).

Suppose an attempt is made to explicitly delete $x1$. According to the ORN semantics as informally described in Section 2, the explicit deletion of $x1$ should result in an implicit destruction of the $y2 \leftrightarrow x1$ instance of $R1$ (or the $x1 \leftrightarrow y2$ instance of $R1^{\sim}$) and the implicit deletion of $y2$. This is based on the $|\sim$ binding and 1 cardinality for class X in the $R1$ relationship. The implicit deletion of $y2$ should result in the implicit destruction of the $y2 \leftrightarrow x1$ instance of $R2$ and the implicit deletion $x1$, based on the $|\sim$ binding and 1 cardinality for class Y in the $R2$ relationship. We have come full circle, but based only on the semantics of Section 2, we should continue by concluding that the implicit deletion of $x1$ should result in the implicit destruction of the $y2 \leftrightarrow x1$ instance of $R1$ and the implicit deletion of $y2$, etc. etc. We are in an infinite loop and have not yet begun to analyze the implicit destructiveness of the other relationships in which $x1$ might be involved—which in Fig. 4 is just $R2$—and their impact on the explicit deletion of $x1$. Obviously, any formal specification of ORN semantics must avoid such circularity of description.

In the formal specification in Section 3, the recursive function definitions detect relationship cycles and avoid circularity by means of their dependence on the set of objects that have already been traversed. To illustrate this, let *dbFig4* represent the state schema for the database shown in Fig. 4, and assume that the operation schema for

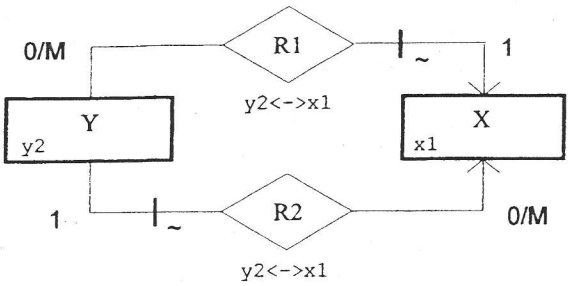



Fig. 4. Relationship cycle $x1 \leftrightarrow y2$, $y2 \leftrightarrow x1$.

*DeleteObject* is invoked to describe the deletion of $x1$. The list below traces the functions invoked by the first precondition in the schema. The last invocation of *rO_implicitly_deletable* returns *true* and recursion terminates since $i.rO \in dos$, i.e. $x1 \in \{x1, y2\}$ and no $i2$s satisfy the $i.rO \notin dos$ constraint in the $\forall i2...$ quantification.

*deletable*(*dbFig4*, x2)
*implicitly_destructible*(*dbFig4*, x1↔y2 of *R1*~, {x1})
*rO_implicitly_deletable*(*dbFig4*, x1↔y2 of *R1*~, {x1})
*implicitly_destructible*(*dbFig4*, y2↔x1 of *R2*, {x1, y2})
*rO_implicitly_deletable*(*dbFig4*, y2↔x1 of *R2*, {x1, y2})

Fig. 5 depicts another relationship cycle and is used to illustrate the second problem that can arise from such cycles. The "?" in the figure indicates the selection of an implicit destructibility binding. For two such bindings we examine what happens when an attempt is made to delete $x1$. We again at first assume only the ORN semantics as described in Section 2.

*Case 1.* "?" is replaced by a default implicit destructibility binding (i.e. no explicit binding indicator is given).

- If $R1$ is considered first, the deletion of $x1$ should result in the implicit destruction of the $y1 \leftrightarrow x1$ instance of $R1$ and an implicit delete on $y1$. This should be successful and result in the implicit destruction of the $y1 \leftrightarrow x1$ instance of $R2$ based on the default implicit destructibility binding for class Y in the $R2$ relationship. Now, when $R2$ is considered to see if any instances involving $x1$ exist that require implicit destruction, it is not clear if any will be found. If in fact the $y1 \leftrightarrow x1$ instance of $R2$ has been implicitly destroyed, the delete of $x1$ should be successful.
- If $R2$ considered before $R1$, the deletion of $x1$ should be unsuccessful because the |- binding prevents the implicit destruction of the $y1 \leftrightarrow x1$ instance of $R2$.

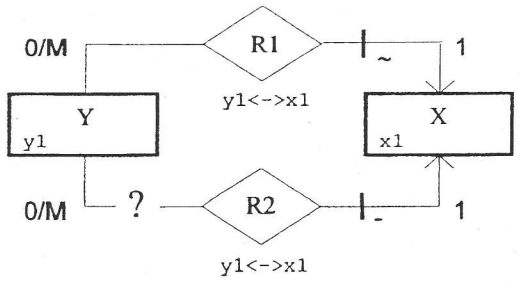



Fig. 5. Relationship cycle $x1 \leftrightarrow y1$, $y1 \leftrightarrow x1$.

The second problem with relationship cycles is evident from the above case. They can cause the outcome of a complex object operation, like object deletion, to be dependent on the order in which relationships are considered (or "processed"). When such ordering is unspecified—as it is in the informal description of ORN semantics of Section 2 and the formal mathematical notations of Section 3, involving iterations over (unordered) sets—ambiguities result. In the "$\forall i$: *db.instances...*" quantification in the function *deletable* in Section 3, which relationship instance will be considered (or in an implementation processed) first?

Now let us assume a different binding for the "?" in Fig. 5, and examine what happens when the an attempt is made to delete $x1$.

*Case 2.* "?" is replaced by a |- implicit destructibility binding.

- If $R1$ is considered first, the deletion of $x1$ should again cause an implicit destruction of the $y1 \leftrightarrow x1$ instance of $R1$ and an implicit delete on $y1$. The deletion of $y1$, however, will not be successful because of the |- binding for Y in the $R2$ relationship. Thus the deletion of $x1$ should be unsuccessful.
- If $R2$ is consider first, the results are the same as indicated in case 1. The deletion of $x1$ should be unsuccessful.

Here the outcome of the operation is independent of the order in which the relationships are processed.

In the formal specification of ORN, we assumed that a one-sided |- binding could not be given for a relationship involved in a relationship cycle. The reason for this restriction was too avoid the ambiguity in semantics exemplified by case 1. Only the |- binding of ORN causes processing order dependencies, and this is true only when it is given for just one class, i.e. one side, in a relationship involved in a relationship cycle. This is indicated by the above two cases and is formally proven in Ref. [13].

We should note that the OR+ implementation of ORN allows the cyclic, one-sided |- binding, which was disallowed in the formal specification. The user, however, is cautioned to avoid it [13]. This problematic binding could have been forbidden but was tolerated for three reasons. First, it could prove useful in defining some relationships. In Fig. 2, for example, the one-sided |- binding occurs for two relationships, though both are non-cyclic. Second, the cyclic nature of the binding cannot be detected at database definition time. The *possibility* of a relationship cycle is detectable when relationships are defined but not an actual occurrence, which is data dependent and not inevitable. And third, when a cyclic, one-sided |- binding does occur, an OR+ user can know (and also indirectly control) the

order in which relationships are processed, thus eliminating the ambiguity exemplified by case 1. In OR+, relationships for an object are always processed in the order in which they are defined within a class.

The algorithms in OR+ implement ORN semantics by implicitly destroying relationship instances as they traverse the database. The database functions in the formal specification, however, do not "process" the database in this manner. That is, they do not (and cannot) change the state of the database by destroying instances as they traverse it (else they would not be true functions). Therefore, in a relationship cycle such as seen in case 1, all instances are eventually examined from the perspective of both sides of the relationship, no matter the order of traversal. This means that a one-sided |- binding encountered in a relationship cycle would always specify the non-deletability of an object, which is not always the result as implemented in OR+. Thus, admittedly, another reason the cyclic, one-sided |- binding is disallowed in the formal specification is the extreme difficulty of formally specifying its messy, processing order dependent semantics as implemented in OR+.

## 5. Conclusion

ORN is a simple yet powerful notation for describing relationship semantics at a very high level of abstraction, the entity(or object)-relationship level. This level of database abstraction is suitable for both system requirements specification and database definition.

This paper has described ORN semantics using formal methods. The formal specification given is precise, unambiguous, and non-circular, accounting for the possibility of relationship cycles within the database. It is also complete in describing ORN minus the cyclic, one-sided |- binding—a situation to be avoided whose meaning is ambiguous in the informal description of ORN and processing order dependent in its implementation. The formal specification of ORN ties the manipulation of objects and relationship instances within a database to the insertion and deletion of objects within sets and of ordered pairs within relations defined on those sets, thus providing ORN with a mathematical interpretation. A mathematically based specification of ORN is beneficial to the potential user and implementer of ORN and facilitates further research into its application and extension.

## Acknowledgements

This work was partially supported by the National Science Foundation under grant CDA-9313299 and cooperative agreement HRD-9707076. Portions of Section 2 are reprinted from Ref. [2] with permission from the publisher, © 1996 ACM 0-89791-826-6. 

## References

[1] W. Kim, Object-oriented databases: definition and research directions, IEEE Trans. Knowledge Data Engng 2 (3) (1990) 327–341.
[2] B.K. Ehlmann, M.A. Stewart, Incorporating Object Relationship Notation (ORN) into SQL, Proc. 35th ACM Southeast Conference, Murfreesboro, TN, April 2–4, 1997, pp. 282–289.
[3] B.K. Ehlmann, G.A. Riccardi, comparison of ORN to other declarative schemes for specifying relationship semantics, Info. Software Technol. 38 (7) (1996) 455–465.
[4] B.K. Ehlmann, G.A. Riccardi, An integrated and enhanced methodology for modeling and implementing object relationships, J. Object-Oriented Programming 10 (2) (1997) 47–55.
[5] P.P. Chen, The entity-relationship model: towards a unified view of data, ACM Trans. Database Systems 1 (1) (1976) 1–36.
[6] B.K. Ehlmann, G.A. Riccardi, Object relator plus: a practical tool for developing enhanced object Databases, 13th International Conference on Data Engineering, Birmingham, England, April 7–11, 1997, pp. 412–421.
[7] T. Atwood, et al., in: R.G.G. Cattel, et al. (Eds.), The Object Database Standard: ODMG-93 Release 1.2, Morgan Kaufmann, San Mateo, CA, 1996.
[8] S.K. Hardeman, Relationship behavior in object databases: subtleties and inconsistencies, student paper (B.K. Ehlmann, advisor), Proceedings of the 34th ACM Southeast Conference, Tuskegee, AL, 1996, pp. 224–229.
[9] S.G. Brown, The ORN simulator: a practical tool for modeling relationship behavior, student paper (B.K. Ehlmann, advisor), Proceedings ADMI'97—Increasing Diversity in Research and Education: The Symposium on Computing at Minority Institutions, ADMI, Washington DC, May 29–June 1, 1997, pp. 130–135.
[10] J.B. Wordsworth, Software Development with Z, Addison-Wesley, Wokingham, UK, 1992.
[11] C.J. Date, Relational Database Writings 1985–1989, Addison-Wesley, Reading, MA, 1990, p. 119–125, 143–147.
[12] B.M. Horowitz, A run-time execution model for referential integrity maintenance, Proceedings of the 8th International Data Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992 pp. 548–556.
[13] B.K. Ehlmann, N. Rishe, J. Shi, Relationship cycles and ORN, submitted for publication.