

Object Relater *Plus*

Object Database Definition Language (ODDL) User's Guide

Release 3.1

Copyright © 1994 1995 1996 1999 2006 2008 Bryon K. Ehlmann

1 Introduction

Object Relater *Plus* (OR+) is a tool for developing object databases (ODBs) [5]. It allows relationships between objects to be more easily implemented and maintained, plus it provides other features, like persistence and ODMG compatibility. OR+ was developed at Florida State and Florida A&M Universities and Southern Illinois University Edwardsville to facilitate the development of scientific databases and tools and to research and experiment with ODB enhancements [2, 3]. Object persistence is provided in OR+ via Object Store [6].

The Object Database Definition Language (ODDL) of OR+ primarily describes an ODB at a conceptual level; however, footnotes provide low-level, implementation details and sidebars provide higher-level, applications related information. The language is called a *database* definition language rather than a data definition language because, being object-oriented, it defines behavior as well as data.

ODDL extends the ODMG standard [1]. While generally compatible with C++ ODL, its relationship declarations are at a higher-level, compatible with generic ODL. The main extension ODDL makes to ODMG-93 is the Object Relationship Notation (ORN) for improved description of aggregate relationship semantics [4]. Generic ODB operations on an ODDL-defined database are provided by the Object Database Manipulation Language (ODML), which is compatible with the ODMG-93, C++ OML standard (future binding). Again, ODML extends this standard to support ORN.

OR+ was originally developed as a research prototype to demonstrate that ORN can be effectively used in developing object databases. Much more information about ORN—for example, its use in database modeling, its integration into relational DBMSs, and the algorithms required for its implementation in both relational and object DBMSs—is extensively discussed in the following book:

B. K. Ehlmann, *Object Relationship Notation (ORN) for Database Applications: Enhancing the Modeling and Implementation of Associations*, Springer Science+Business Media, LLC, New York, NY, 2009, www.springer.com.

The remainder of this document defines the syntax and semantics of ODDL: the primary ODDL language in Section 2, ORN in Section 3, and control commands in Section 4. It also describes the Unix command required to execute ODDL in Section 5. An Appendix lists some language features not yet fully implemented.

2 Syntax and Semantics

ODDL syntax is described by the syntax diagrams in Figs. 1 and 2. Semantics are provided by the descriptiveness of the syntax and by the explanations below. Fig. 3 gives example ODDL specifications for part of a company database.

ODDL-spec

Fig. 1 shows that an *ODDL-spec*, or ODDL specification, consists of a number of forward class declarations, class definitions, and extent definitions. An extent is a collection containing all objects of a particular class. C++ code is interspersed within an *ODDL-spec*, via control commands, to provide for the specification of user-defined attribute types, private attributes, and object behavior in the form of function member definitions or signatures. Other control commands interspersed within an *ODDL-spec* provide sidebars and footnotes. (See Section 4.)

Comments given in ODDL specifications at places prescribed by the syntax diagrams are associated with the appropriate database *meta-object*—the database, a class, an extent, or an attribute—and stored in the ODDL meta-database.

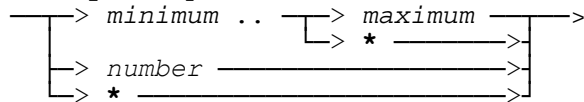
Refs are sidebar and/or footnote references. Sidebars and footnotes provide additional specifications related to a meta-object. (See **\$sidebar** and **\$footnotes** in the Section 4.)

class-declaration

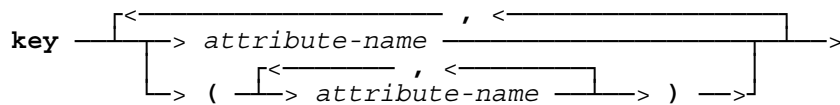
A *class-declaration* is a forward declaration of a class-name. It allows a class to be used in defining another class before it itself is defined.

4/23/09

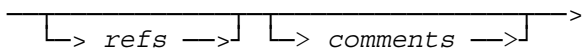
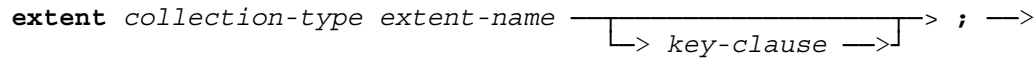
multiplicity:



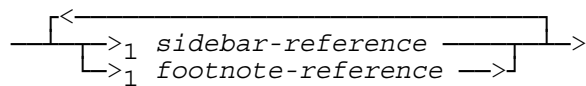
key-clause:



extent:



refs:



sidebar-reference:



footnote-reference:



comments:

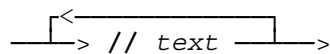
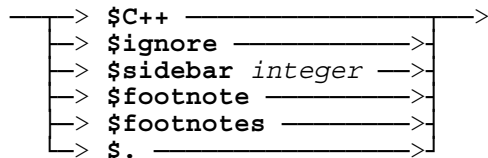
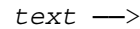


Figure 1. ODDL Syntax Diagrams (continued)

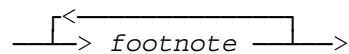
control-command:



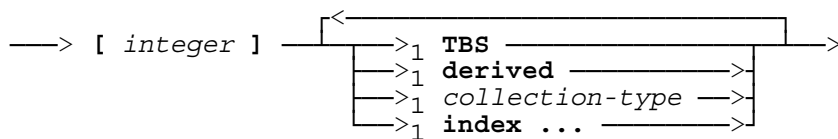
sidebar:



footnotes:



footnote:



collection-type:

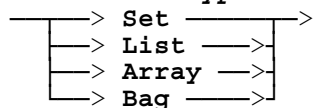


Figure 2. ODDL Control Command Syntax Diagrams

```

Database CompanyDB // Company DataBase

$sidebar 1
Identifier // Generated applications code will ensure uniqueness of attribute values.
$.
$C++
class date {...};
typedef int building_no;
...
$.
{
  class address;
  class payroll_record;
  class department;
  class position;
  class assignment;
  class dependent;

  class employee {
    d_String      SSN; |1 // Social Security Number
    d_String      Name; // Last name, First name
    date          BirthDate;
    address        Address inverse Addressee '<*-to-0..1>|-';
    payroll_record Payroll inverse Employee '!X-<1-to-1>X-';
    department     Dept inverse Employees '<*-to-1>';
    position       Position; // Relationship semantics not maintained by system.
    Set<assignment> Tasks inverse Employee '!<1-to-*>';
    employee       Spouse inverse Spouse; // association defaults to <0..1-to-0..1>;
    List<dependent> Dependents '!<1-to-*>';
    car_pool       CarPool inverse Riders '|?X?<2..*-to-0..1>';
$C++
    void RaiseSalary(int percentage);
    void CancelAssignments();
    ...
$.
  };
  extent Collection<employee> Employees key SSN; [1]

  class department {
    d_String      Name; |1
    building_no   Location; // Building No.
    Set<employee> Employees inverse Dept '<1-to-*>';
    ...
  };
  extent Set<department> Departments key Name;

  class assignment {
    project       Project inverse Assignments '<*-to-1>!';
    employee       Employee inverse Tasks '<*-to-1>!';
    date          StartingDate;
    date          TerminationDate;
    int           Duration; [2] // in days
$C++
    void Extend(int days); // Extend assignment.
    ...
$.
  };
  extent Set<assignment> Assignments key (Project, Employee);
...
};
$footnotes
[1] Set
[2] derived
...
$.

```

Figure 3. ODDL Example

class

A *class* defines a class, which is an object type whose instances have an identity independent of their attribute values. The **isa**, or **: public**, clause specifies “is a” relationships. A *class-name* given in an **isa** clause must identify a class defined elsewhere in the ODDL specification. C++ inheritance is operative for the **isa** relationship.

An **X-** indicates that objects of the class are not explicitly deletable. Nonexplicit deletability is inherited by the subclasses of a class.

An *attribute* specifies an attribute common to all objects of the class that serves to define or describe the objects.

attribute

An *attribute* is either *literal-valued* or *object-valued*. A *literal-valued-attribute* stores values that have no independent identity in the database—e.g., *BirthDate* in the class *employee* (see Fig. 3). An *object-valued-attribute* references objects in the database that have identities independent of the values that describe them—e.g., *Dependents* for class *employee* in Fig. 3.

literal-valued-attribute

The *literal-type* in a *literal-valued-attribute* is a C++ primitive type, e.g., an **int**; an ODDL-defined type, e.g., *d_String*; or a previously defined C++ user-defined type, e.g., an enum, typedef, or class, or C++ macro. The type cannot be an user declared ODDL class. Operations that can be performed on a literal-valued attribute are completely determined by its type.

object-valued-attribute

An *object-valued-attribute* of type *class-name* (or *class-name** in C++ terms) defines a single-valued attribute whose value is **NULL** or references an object of the class identified by *class-name*. The attribute represents the “to one” part of a “one to one” or “many to one” relationship between the class containing the attribute, the *subject class*, and the class being referenced, the *relative class*.

An attribute of type *collection-type* defines a multi-valued attribute whose values are a collection of references to objects of the class identified by the *class-name* given in the *collection-type*. The attribute represents the “to many” part of a “one to many” or “many to many” relationship between the class containing the attribute, the subject class, and the class being referenced, the relative class.

An object-valued *attribute-name* should indicate the role that the referenced class object or objects play in describing the subject class. Often, the *attribute-name* is the same as the referenced *class-name* except for differences in case, upper and lower, or number, singular and plural. ODDL is case sensitive.

The **inverse** clause indicates that the relationship represented by the object-valued attribute is to be maintained by the database system. When given, the keyword **relationship** can be also given for ODMG-93 ODL compatibility. The *attribute-name* given in inverse clause identifies the **inverse attribute**, the attribute that represents the inverse relationship. Inverse attributes permit the relationship to be meaningfully viewed from the context of the relative class, thus facilitating the specification of queries from this context. They also provide for efficient access of subject class objects from relative class objects. Inverse attributes are automatically maintained by the system in the face of object deletions and relationship creations, destructions, and changes. The inverse *attribute-name* must be the name of an appropriate object-valued attribute of the relative class. An object-valued attribute and its inverse must cross reference each other with inverse clauses.

An *association* describes in Object Relationship Notation (ORN) the semantics of relationship—or more precisely, an association, represented by the object-valued attribute and its inverse. ORN is discussed more fully in Section 3. An association need only be given for one attribute of an attribute/inverse attribute pair. The attribute without the association implicitly assumes the inverse of the given *association*—e.g., $|-<0..1\text{-to-}1..*>$ is the inverse of $<1..*\text{-to-}0..1>|-$. If a *association* is given for both attributes of an attribute/inverse attribute pair, the *association* given for one must be the inverse of the one given for the other. If a *association* is not given for both an attribute and its inverse, a default *association* is assigned based on whether the attributes are single-valued or multi-valued. For a single-valued attribute, the default *association* is $<... \text{-to-} 0..1>$. For a multi-valued attribute, it is $<... \text{-to-} *>$. The *multiplicities* for a single-valued, object-based attribute, if given, must be $\text{-to-} 0..1$ or $\text{-to-} 1$. If a $\text{-to-} 1$ association is specified, the attribute value cannot be **NULL**. If a $\text{-to-} 1..*$ (or a $\text{-to-} n..*$ or $\text{-to-} n..m$ where $n > 0$) relationship is specified for a multi-valued, object-valued attribute, the attribute cannot be empty.

Operations that can be performed on any object-valued attribute are to create, destroy, and change an association of the type identified by the attribute. Basically, for a single-valued attribute, a create is done when a **NULL** reference is set to a non-**NULL** reference, a destroy is done when a non-**NULL** reference is set to **NULL**, and a change is done when a non-**NULL** reference is set to another non-**NULL** reference. For a multi-valued attribute, a create is done when a reference is inserted into the attribute, a destroy is done when a reference is removed, and a change is done when a reference in the attribute is replaced with another reference. These operations implement the association semantics as prescribed by the *association* and **inverse** clause via multiplicity checks and side-effects.

The *key-clause* specifies one or more keys for the objects within a multi-valued, object-valued attribute. A key is an attribute or group of attributes of the relative class that uniquely identify the class objects referenced by the attribute. (See *key-clause*.)

collection-type

An *collection-type* defines a collection of object references to objects of type identified by the *class-name*. Generic operations on a collection are to check if it is empty, to check if it contains a particular object reference, to insert and

remove an object reference, and to sequentially access all objects referenced via an iterator. Other operations depend on the specific type of collection specified. Retrieval of collection members in a specific order or random retrieval via an index can be added and supported by specifying a particular type of collection or giving additional specifications in a footnote. See Section 4 for additional details on collection types.

association

See Section 3.

key-clause

A *key-clause* specifies one or more keys for the objects referenced in a collection. A key is an attribute or group of attributes of the relative class that uniquely identify the class objects referenced in the collection. An *attribute-name* must reference a single-valued attribute. The (and) delimit attributes that only in combination uniquely identify an object within the collection, i.e., a *compound key*. Object-valued attributes declared as a key or part of a key cannot be **NULL**, i.e., their associated *multiplicities* must be **-to-1** (See Section 3.)

extent

An *extent* defines a class *extent*, which is a collection that includes all objects of a class. The class is identified by the *class-name* specified in the *extent*. Objects of the class are inserted into the extent when they are created and removed from the extent when they are deleted. Specifying an extent for a class allows all objects of the class to be sequentially and directly accessible. Random access to objects in an extent is supported by associating index structures with the extent via a footnote. (See Section 4.)

refs

Refs are references to a sidebar, a footnote, or both. In a syntax diagram, like that for *refs*, an arrow subscripted by “1” means the directed path can only be taken once for each entry into the diagram.

A *sidebar-reference* references a sidebar. The given *integer* should match the number of a previously specified sidebar (see **\$sidebar** in Section 4).

A *footnote-reference* references a footnote. The given *integer* should match an *integer* given in a subsequent footnote (see **\$footnote** in Section 4). Once an integer has been used to reference a footnote and the footnote has been given, the integer can be reused.

comments

Comments consist of one or more lines of comment. The double slash identifies the beginning of *comments* and its continuation on a new line. *text* is all characters between a double slash and the end of line.

3 Relationships

ORN (Object Relationship Notation) allows relationship semantics to be specified to an ODBMS when relationships are represented as object-valued class attributes, both single-valued and multi-valued. These relationships are of the type termed *associations* in UML [7]. ORN is compatible to the syntax used to specify association semantics in UML. The syntax of a *association* corresponds to ORN and is given in Fig. 1. The semantics are given in Table 1. The remainder of this section provides a detailed description of ORN.

Table 1. ORN Semantics

Semantics are given in terms of a subject class *S* with multiplicity *m* and binding *b* in an association *A* with a related class *R* (which could be *S* in a different role).

A is implemented by an object-based attribute *roleR* and inverse object-based attribute *roleS*. The type of *roleR* is *R* or some suitable collection of *R* (shown here as *R*), and the type of *roleS* is *S* or some suitable collection of *S* (shown here as *Set<S>*). The multiplicity for *S* (or *roleR*) is *m*, and the binding is *b*. An *A* link, i.e. an “*A* reference pair,” is destroyed when a *roleS* or *roleR* reference is destroyed or changed to reference a new object. Creating or destroying either reference is equivalent to and includes creating and destroying, respectively, its inverse reference.

multiplicity: Semantics are the same as those in UML. Essentially, *m* indicates a lower bound and upper bound on the number of objects of type *S* that can be related via *A* to each object of type *R*.

An *R* object can be created provided this does not violate *m*. The check for a lower bound violation is deferred until transaction commit. A *roleR* reference can be created provided this does not violate *m*. The check for an upper bound violation is immediate. The enforcement of *m* on the deletion of an *S* object or destruction of a *roleR* (or *roleS*) reference is determined by the binding *b*.

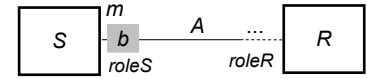
binding: *A* | in *b* denotes a “cut” and an Implicit, i.e., system initiated, destruction of an existing *A* reference that must occur on deletion of an *S* object. An *X* in *b* denotes a “cross out” and an explicit, user initiated, destruction of an *A* reference.¹

An *S* object deletion and an explicit *A* reference destruction are *complex object operations*. Deletion of an *S* object succeeds only if all existing association references involving that object are implicitly destructible. Also, deletion of an *S* object or destruction of an *A* reference succeeds only if all required implicit object deletions succeed.

di: A destructibility indicator in *b* specifies the destructibility of an *A* (i.e., *roleR* and inverse *roleS*) reference. The meaning of each indicator is given below. This meaning can alternatively be described by the actions taken on an attempt to destroy an *A* reference. These actions are given in brackets. If a *di* is given after a |, it applies to implicit reference destruction; if given after an *X*, it applies to explicit reference destruction; and if given alone, it applies to both. If a *di* is not given, i.e., is nil, for implicit reference destruction, explicit reference destruction, or both, default destructibility applies to whichever.

- nil *Default destructibility.* A reference can be destroyed provided this does not violate *m*.² [Destroy the reference. If *m* is violated², raise an exception³.]
- *Negative destructibility.* A reference cannot be destroyed. [Raise an exception.³]
- ~? or ? *Conditional cascade destructibility.* A reference can be destroyed, but if this violates *m* (?), the destruction must be cascaded (~) to the related *R* object, i.e., this object must be implicitly deleted. [Destroy the reference. Delete the related *R* object? If *m* is violated, yes; else no.]
- ~! or ! *Emphatic cascade destructibility.* A reference can be destroyed, but the destruction must be cascaded (~) to the related *R* object. [Destroy the reference. Delete the related *R* object!]
- ~' or ' *Tentative (or qualified) cascade destructibility.* A reference can be destroyed, but an attempt must be made to cascade (~) the destruction to the related *R* object; however, this implicit *R* object deletion must be undone if it fails, but is required if and only if its undoing would violate *m*.² (Think of the ' as a “pruned back !” or as a “qualifying footnote reference” on the cascade.) [Destroy the reference. Delete the related *R* object.' (' – If an exception occurs, undo the delete and then, if *m* is violated², raise an exception³.)]

1 - A *roleS* reference change done as a single operation that replaces an *S* object with another is not treated as an explicit link destruction relative to class *S* (but is relative to *R*) and is allowed if allowed by other multiplicities and bindings.
 2 - The check for a lower bound violation is deferred until the end of the current complex object operation.
 3 - The current complex object operation is undone.



```
class S {
    ...
    R roleR inverse roleS b<m-to-...;
}

class R {
    ...
    Set<S> roleS inverse roleR;
}
```

An *association* describes associations at two levels: first in terms of object multiplicities and then in terms of object bindings. The *multiplicities* define an association type solely by subject and relative class *multiplicities*. *Bindings* are then added in an *association* to indicate the level of *binding* between related objects, i.e., the destructibility of association instances and the implicit deletability of objects based on the association.

3.1 Multiplicities

In *multiplicities*, the *multiplicity* before the -to- describes the multiplicity for the subject class, the class containing the object-valued attribute for which an association is being specified. The *multiplicity* after the -to- describes the multiplicity for the relative class, the class whose objects are being referenced by the object-valued attribute. The relative class multiplicity indicates the number of objects of the relative class that can relate to a single object of the subject class. Likewise, the subject class multiplicity indicates the number of objects of the subject class that can relate to a single object of the relative class. A multiplicity is expressed as a lower and upper bound, where .. denotes “to” and * denotes “infinity.” When used by itself, * denotes “zero to infinity.”

The symbols .., *, 0, and 1 are adequate to describe the multiplicities of most associations. As an example, the association between departments and employees could be described as 1-to-*. Each department, the subject class object, relates to zero, one, or more employees. Each employee, the relative class object, relates to exactly one department.

Although not required to describe most associations, the integer..integer notation provides for multiplicities with lower and upper bounds other than 0, 1, and *. For example, 2..* means two or more, and 2..10 means between two to ten, inclusive. When the multiplicity is given as a single integer, that integer provides both the lower and upper

bounds, e.g., a multiplicity of 6 is equivalent to 6..6. For the most part, the discussion of associations in the remainder of this section is based on associations having multiplicities expressed only in terms of 0, 1, and *.

An *intra-class association* occurs when the subject and relative class are the same class. An intra-class *multiplicities* cannot be 0..1-to-1, 0..1-to-1..*, or 1-to-1..*. These are mathematically impossible, as shown in [3], though they can occur between subclasses of a class, which are discussed in Section 4.

The semantics prescribed for an association type, hereafter referred to as *association semantics*, place constraints on the state of the database and specify that certain database operations perform checks and have desirable side effects in order to maintain these constraints. Affected operations are:

- add a class object,
- delete a class object,
- create an association link between two class objects,
- destroy an association link between two class objects, and
- change an association link for a class object so it relates to a different class object.

These operations are at a higher level than similar object-level operations that are typical in existing ODB systems. All constraints imposed by association semantics must be satisfied at the time a database transaction commits.

Association semantics as defined by *multiplicities* are derived from the semantics of the multiplicities for the subject class and the relative class. For example, the semantics of a 1-to-* association type are the semantics of a -to-* specification plus the semantics of a 1-to- specification. *Multiplicity semantics* are given below in terms of an association *A* having the given multiplicity for the relative class. The semantics in terms of an inverse association having the multiplicity for the subject class are obtained by substituting the terms “relative” for “subject” and “subject” for “relative” wherever they occur below.

-to-1 (inverse 1-to-). Each subject class object depends on a relative class object for its existence. A subject object cannot be added without having an *A* association with a relative object. A subject object must be deleted if deletion of its relative object is permitted or destruction of its *A* association (either directly or by an association change for the relative object) is permitted.

For example, a *-to-1 association between employees and departments means that each employee must have a department. Deletion of an employee's department or destruction of its association with a department, if permitted, would require deletion of the employee.

-to-1..* (inverse 1..*-to-). Each subject class object depends on one or more relative class objects for its existence. A subject object cannot be added without having an *A* association with at least one relative object. A subject object must be deleted if deletion of its only relative object or destruction of its only *A* association is permitted.

-to-0..1 (inverse 0..1-to-). A subject class object does not depend on a relative class object for its existence. Deletion of a subject object is never required by the multiplicity of its relative class.

-to-* (inverse *-to-). A subject class object does not depend on a relative class object for its existence. Deletion of a subject object is never required by the multiplicity of its relative class.

For specifications -to-1..*, -to-0..1, and -to-*, an *A* association that a subject object has with a relative object must be implicitly destroyed if deletion of the relative object is permitted and this deletion does not require the deletion of the subject object (as is possible in -to-1..*).

The four possible multiplicities—1, 1..*, 0..1, and 0..*—for the two classes in an association result in ten distinct and basic association types. Six of the sixteen possible combinations are indistinct since associations are bi-directional. Association semantics are more finely tuned and more association types result when a *multiplicities* description is extended to an *association*.

3.2 An association

A *association* provides a more detailed level of association type definition by indicating the level of subject and relative class *binding*. Bindings define the implicit and explicit destructibility of associations and whether association destruction can result in the implicit deletion of related objects. An association, or link, between objects is implemented in OR+ as a reference and corresponding inverse reference. All existing links involving (references from/to) an object must be implicitly destroyed before an object can be deleted. Implicit deletions resulting from object deletion and link destruction enforce multiplicities and can define the extent of a *composite object*. This in turn can define the meanings of other composite object operations—e.g., copy and print. A composite print on an object x would print x as well as all objects whose deletion would be attempted as a result of the deletion of x .

The *binding* before the $<$ in an *association* indicates the binding for the subject class; the one after the $>$ indicates the binding for the relative class. Association semantics as defined by an *association* are derived from multiplicity semantics and the semantics of the specified bindings.

Default Binding. The default binding for a subject class S in an association A denotes that existing A links are both implicitly destructible on deletion of an S object and explicitly destructible, provided this destruction does not violate the multiplicity for S in A . That is, when an S object is deleted, implicitly or explicitly, each A link it has with another object is implicitly cut, i.e., the references and inverse references are destroyed. If this destruction results in a violation of the multiplicity for S in A , the delete is disallowed. Also, an explicit destruction of an A link (either directly or via an association change) is allowed if and only if the multiplicity for S in A is not violated. Checks for multiplicity violations are made when the complex object operation involving the delete or explicit link destruction is committed. For example, the default binding for the employee class in a $<*-to-1>$ association between employees and departments denotes that an existing link between an employee and a department is implicitly destroyed when the employee is deleted. Such destruction never violates the $*$ multiplicity for the employee class. The default binding for the department class in the association denotes that a department cannot be deleted if it has any employees since implicit link destruction would violate the 1 multiplicity for the department class, i.e., an employee must have 1 department. Also, a link between an employee and a department cannot be explicitly destroyed as this would violate the 1 multiplicity, though an employee's department may be changed.

Implicit Destructibility Bindings. The $| -$ (minus) implicit destructibility binding for a class S in an association A denotes that existing A references are never implicitly destructible, i.e., existing links can never be cut, on deletion of an S object. That is, deletion of an S object fails when it has an A reference with another object. For example, if the association between employees and departments was $| -<*-to-0..1>$, then an employee (here, the S object) could exist without being in a department; however, if in a department, the employee cannot be deleted. A $| -$ binding is superfluous for a 1 multiplicity since implicit destruction of an existing A reference always violates the multiplicity and fails with the default binding.

The $| \sim ?$ or $| ?$ implicit destructibility binding for a class S in an association A is only meaningful for multiplicities with lower bound > 0 and denotes that existing A links are always implicitly destructible, i.e., can always be cut, on deletion of an S object. When implicit destruction violates the multiplicity for S in A , the related object is implicitly deleted—i.e., deletion effectively propagates like a wave (\sim). If deletion of the related object fails (because of constraints involving this or another association), deletion of the S object fails. For example, if the association between employees and departments was $| ?<1..*-to-1>$, then deletion of the last employee (here again the S object) in a department would cause the deletion of the department. If the association was $| ?<1..* -to-1>| ?$, then deletion of a department (now viewed as the S object) would cause deletion of all employees in the department.

The $| '$ binding for a class S in an association A denotes that when an S object is deleted, each A link (reference and inverse reference) it has with a subordinate object is implicitly destroyed, and an implicit delete is attempted on the related object. Failure of an implicit delete on a related object will result in its undoing but cause the failure of an S object delete if and only if the implicit deletion is needed to maintain the multiplicity for S in A . To illustrate, if the association between departments and employees was $| '<1-to-*>$, then when a department is deleted, all of links 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 multiplicity constraint must be maintained. If, however, the association was instead $| '<0..1-to-*>$, then the department delete would not fail if a related employee delete failed. The related employee would simply remain (the implicit delete having been undone).

The $| !$ binding for a class S in an association A is similar to the $| '$ binding except that the implicit delete that is done on the related object must always succeed. The $| !$ binding is similar to the $| ?$ binding except that the implicit delete is unconditional. To illustrate, if the association between departments and employees was $| !<0..1-to-*>$, then when a department is deleted, all of the associations 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 (even though the $0..1$ multiplicity constraint is not violated).

Explicit Destructibility Bindings. The **x-** binding for a class *S* in an association *A* denotes that *A* links are not explicitly destructible. For example, if the association between employees and departments was **x-<*-to-0..1>**, then a link between an employee and a department once created cannot be explicitly destroyed. Note, however, that it can be “indirectly” destroyed by an atomic change operation that replaces the employee in the link with another employee. See footnote 1 in Table 1. (It can also, of course, be implicitly destroyed if the employee is deleted.) When the **x-** binding is given for both ends of an association, e.g., **x-<*-to-0..1>x-**, it effectively binds objects “until death do ye part.”

The **x?** binding for a class *S* in an association *A* is only meaningful for multiplicities with lower bound > 0 and denotes that *A* links are always explicitly destructible. When explicit destruction violates the multiplicity for *S* in *A*, the related object is implicitly deleted. If the deletion of the related object fails, the explicit link destruction fails. For example, if the association between employees and departments was **x?<1..*-to-0..1>**, then explicit destruction of a link between a department and the last employee in the department would cause the deletion of the department. If the association was **x?<1..*-to-1>x?**, then explicit destruction of a department and employee link 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!

The **x'** binding for a class *S* in an association *A* denotes that when an *A* link is explicitly destroyed, an implicit delete is attempted on the related object. Failure of this implicit delete will cause the failure of the explicit *A* link destruction if and only if the implicit deletion is needed to maintain the multiplicity for *S* in *A*. To illustrate, if the association between departments and employees was **x'<1-to-*>**, then an explicit destruction of a department and employee link would cause the attempted deletion of the employee. If this delete fails, the department delete fails since the **1** multiplicity constraint must be maintained. If the association was instead **x'<0..1-to-*>**, then the explicit link destruction would not fail if the employee delete failed. The employee would simply remain (the implicit delete having been undone).

The **x!** binding for a class *S* in an association *A* is similar to the **x'** binding except that the implicit delete that is done on the related object must always succeed. The **x!** binding is similar to the **x?** binding except that the implicit delete is unconditional. To illustrate, if the association between departments and employees was **x!<0..1-to-*>**, then an explicit destruction of a department and employee link would cause the attempted deletion of the employee. If this delete fails, the explicit link destruction fails (even though the **0..1** multiplicity constraint is not violated).

4 Control Commands

Control commands in ODDL provide a number of auxiliary functions. They permit the specification of attribute types and object behavior to be given directly in C++. They also allow the specification of sidebars and footnotes, whose functions are described below. Finally, control commands permit selective processing of ODDL specifications.

Fig. 2 provides control command syntax. A *control-command* must be given on a separate line with the **\$** in column one.

\$C++

\$C++ indicates that C++ code follows on subsequent lines.

C++ code given prior to the “{” of the *ODDL-spec* specifies global, user-defined attribute types, i.e. **enum**, **typedef**, **struct**, and **class** definitions. These must define fixed-size data objects that contain no C++ pointers.

C++ code given within the “{...}” of the *ODDL-spec* prior to a *class* specifies user-defined attribute types, i.e. **enum**, **typedef**, **struct**, and **class** definitions, that are “local” to the subsequent class. These types must again define fixed-sized data objects that contain no pointers.

C++ code given within the “{...}” of a *class* specifies the signatures for member functions. This code can also declare private data members.

\$ignore

\$ignore indicates that subsequent lines should not be processed. A **\$.** terminates the **\$ignore**.

\$sidebar

\$sidebar indicates that subsequent lines are the *text* of a sidebar. A **\$.** control command terminates the sidebar. Sidebars provide high-level application related information that extends the conceptual description of the ODB provide by ODDL. This high-level information may be options relevant to a specific application or to generic applications, e.g. scientific databases. Sidebars are given in a language designed by the OR+ tool user and parsed in `gen.C` prior to the generation of applications code. The class `sb_source`, defined in `source.h`, provides a scanner for sidebars.

The *integer* given is the sidebar number. Once a sidebar is defined, it can be referenced by subsequent meta-objects by giving its number in a *sidebar-reference*.

\$footnote and \$footnotes

\$footnote(s) indicates that *footnotes* follow on subsequent lines. A **\$.** control command terminates the footnotes. Footnotes provide low-level implementation details including internal schema that defines the ODB storage structures for collections and support for random access via keys and sequential access via prescribed orderings.

The *integer* in a *footnote* must match the integer of at least one previous *footnote-reference* given for a meta-object. A *footnote* provides one or more implementation notes relevant to one or more meta-objects.

The footnote **TBS** is an no-op footnote meaning To Be Specified.

The footnote **derived** indicates that a literal-valued attribute is derived.

A *collection-type* specifies that a specific type of collection is to be used for a multi-valued, object-valued attribute or extent that was defined with the generic type **Collection**. The default collection type is **Set**. A **Set** is unordered and allows no duplicate members. A **List** is ordered and allows duplicate members by default. A multi-valued, object-valued attribute with an **inverse** clause, however, does not allow duplicates when defined as a **List**. Also, such an attribute cannot be a **Array** or **Bag**, and two such attributes that are inverses of one another cannot both be **Lists**. See ODMG-93, Release 1.2, documentation for more detailed definitions of **d_Set**, **d_List**, **d_Array**, and **d_Bag**.

The footnote **index** specifies an index on a collection. The exact syntax and semantics of this footnote have not been developed.

5 ODDL Command

The description of the Unix command for executing **oddl** is given in Fig. 4 in man pages format.

The processing of an ODDL specification by **oddl** verifies proper syntax and semantics, checks for inconsistencies in the bindings and multiplicities for associations, and assigns default *association* specifications as required. An error-free ODDL specification can result in the generation of C++/Object Store code, which implements ODML when compiled and linked with database-independent ODML object files. (See ODML User's Guide.) The metadata database built by ODDL can be saved as an Object Store database and later processed by ODDL, the Object Store browser, or another program written by the OR+ tool user.

The **-g** option of the **oddl** command and a number of ODDL supplied "tools" facilitate the extension of ODDL into an even more enhanced ODBMS. When this option is given, the OR+ user assumes control of ODDL code generation. The tools supplied to the user for extending ODDL are a scanner, hooks in the metadata database, a code generation facility, public member functions and iterators that facilitate access to metadata database objects, and a class representing the ODDL provided database system layer.

As previously indicated in Section 4, the scanner is provided for parsing sidebars. The sidebars are initially stored in the metadata database as text. The first step in generating an enhanced ODB system is to parse these sidebars, appropriately issuing errors, if any, and updating an extended ODDL metadata database. Metadata database extension is facilitated by hooks, i.e., extension pointers, attached to each metaobject.

NAME
 oddl - Object Database Definition Language Processor or Object Relater Plus (OR+)

SYNOPSIS
 oddl [-d] [-DBS] [-g] [-p] [-s] [-v] [-mDB database-name | < spec-file]

DESCRIPTION
 oddl processes an ODDL specification within the standard input file, spec-file, creating an in-memory metadatabase or processes a previously saved ODDL metadatabase for the given database-name. Output is determined by specified options. Error messages are written to the standard error file.

OPTIONS

- d Debug ODDL processor printing to the standard output each token in the ODDL specification as it is parsed.
- DBS
 Generate code for default database system. ODDL generates the files <dbn>OR+.h, <dbn>OR+.C, <dbn>OsSchemaOR+.C, and <dbn>OsCollDeclsOR+.h where <dbn> is the ODDL database-name. In addition, if the -hCs option is specified, a <cn>OR+.h and <cn>OR+.C file are generated for each class in the database where <cn> is the class-name. All of these files, in conjunction with certain ODML object files, allow applications code to access and manipulate the ODDL-defined database via ODML. A Makefile is provided that can be edited for proper compilation of applications programs.

 If -hCs is not specified, the <dbn>OR+.C file can be compiled seperately or included in a <dbn>.C file that is written by the OR+ user and contains member function definitions for all database classes. The <dbn>OR+.C file includes the <dbn>OR+.h file. The <dbn>OR+.h file must be included at the beginning of all user-written database application programs, which must be linked to <dbn>OR+.o (and/or <dbn>.o) and the odmlrt library.

 If -hCs is specified, the <dbn>OR+.C file is normally compiled seperately but can be included in a <dbn>.C file that is written by the OR+ user and contains function definitions that are global to the database. Each <cn>OR+.C file can be compiled seperately but is normally included in a <cn>.C file that is written by the OR+ user and contains member function definitions for the specific class. The <cn>OR+.C file includes the <cn>OR+.h file.

 A <cn>OR+.h file must be included at the beginning of a user-written database application program when the program references the specific class. An application program must be linked to <dbn>OR+.o (and/or <dbn>.o), <cn>.o's (and/or <cn>OR+.o's) for all referenced classes, and the odmlrt library.
- g Generate applications code based on metadatabase and sidebars. Generation is based on code written by the OR+ tool user in gen.C.
- hCs
 Generate class-specific code for default database system into <cn>OR+.h and <cn>OR+.C files, where <cn> is a class-name. This option is only applicable if -DBS is specified. See -DBS option.
- mDB database-name
 Process previously saved ODDL metadatabase for database database-name, i.e., process the Object Store database <dbn>OR+MetaDb and the associated code library <dbn>OR+MetaDbCode where <dbn> is the given database-name.
- p Print the ODDL specification to standard output based on the contents of the metadatabase.
- s Save metadatabase as <dbn>OR+MetaDb using Object Store. This database can be read by another program by including the source file metadb.h, linking with metadb.o, and calling metadb::Get (see description in metadb.h). Cannot be given with -mDB option.
- v Output the version of ODDL executed to standard error.

Fig. 4. man Description of Command for Executing ODDL.

A C++ preprocessor, `c^pp`, is provided to facilitate code generation. This tool is illustrated by the code below, which generates for each extent in the metadatabase an `extern` pointer declaration in a `.h` file and a corresponding definition in a `.C` file. `^` (insert) commands are translated to `couts` by `c^pp`.

```
list_cursor e(Mdb->Extents); // Mdb points to the metadatabase
extent* ep;
while (e.More()) {
    ep = (extent*)e.Next();
    SetCodeFile(*Hfile);
^extern d_`ep->ICT`<`ep->ExtendedClass->Name`*>* `ep->Name`_Ptr;/
    SetCodeFile(*Cfile);
^    d_`ep->ICT`<`ep->ExtendedClass->Name`*>*      `ep->Name`_Ptr;/
}
```

For the extent `Employees` in Fig. 3, the above code generates the following pointer declaration in some `.h` file and definition in some `.C` file:

```
extern Set<employees*>* Employees_Ptr;
Set<employees*>*      Employees_Ptr;
```

The public member functions of a class `dbslayer` enable the OR+ tool user to generate all portions of the ODDL provided database system layer. This is the layer of software that the tool user is extending and essentially includes ODMG-93 capabilities enhanced with ORN. The member functions of `dbslayer` give the tool user the flexibility to generate portions of the database system layer into user-specified source files and to mix in user generated code at strategic points.

The ODDL system has been compiled with multiple C++ compilers and run on multiple UNIX platforms. Given the prototype nature of the system, implementation is sometimes constrained by the user capabilities of the commercial ODBMS upon which it is built. In addition, some ODBMS capabilities are only available at the level of its underlying ODMBS, e.g., database schema migration. Obviously, overall implementation could be improved if all capabilities were well integrated into a real ODBMS.

References

1. R.G.G. Cattel, et al., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco, CA, 1997 (see Release 1.1 for description of future binding).
2. B.K. Ehlmann, G.A. Riccardi, and L.C. Dennis, "Representing Non-Inheritance Relationships in an Object-Oriented, Scientific Database," *Proceedings of the Sixth International Working Conference on Scientific and Statistical Database Management*, ETH Zurich, June 1992, pp. 99-109.
3. B.K. Ehlmann, L.C. Dennis, and G.A. Riccardi, "An Object-based Conceptual Model of a Nuclear Physics Experiments Database," *Nuclear Instruments & Methods in Physics Research*, Sect. A, Elsevier Science, North-Holland, **A325**, 1993, 294-308.
4. B.K. Ehlmann and G.A. Riccardi, "A Notation for Describing Aggregate Relationships in an Object-Oriented Data Model," *Applications of Databases—First International Conference Proceedings*, W. Litwin and T. Risch (Eds.), Lecture Notes in Computer Science 819, Springer-Verlag, 1994, pp. 62-77.
5. B.K. Ehlmann and G.A. Riccardi, "Object Relater Plus: A Practical Tool for Developing Enhanced Object Databases," *Proceedings 13th International Conference on Data Engineering*, Birmingham, UK, April 7-11, 1997, IEEE Computer Society Press, Los Alamitos, CA, pp. 412-421.
6. Progress Software Inc., ObjectStore interprise, 2009, www.objectstore.com/datasheet/index.ssp.
7. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.

Appendix

The following syntax, although documented in this guide, is currently not implemented in ODDL:

- the **Array**, and **Bag** footnotes
- the **index** footnote

The following ODDL features are currently not supported by ODML:

- the *key-clause*
- the = *identifier* clause for a *literal-valued-attribute*
- the **derived** footnote