Chapter 3

Object Definition Language

This is not a standard. It is a work-in-progress draft of ODMG's Object Model which incorporates changes which have been approved by the ODMG Board for version 2.0. Additionally, we are working on a proposal to resolve the ambiguities of IDL and ODL interfaces with respect to representation of object state. This draft will expire in January 1997 at which time it will be replaced with an updated version.

3.1 Introduction

The Object Definition Language (ODL) is a specification language used to define the interfaces to object types that conform to the ODMG Object Model. The primary objective of the ODL is to facilitate portability of database schemas across conforming ODBMSs. ODL also provides a step toward interoperability of ODBMSs from multiple vendors.

Several principles have guided the development of the ODL, including

- ODL should support all semantic constructs of the ODMG Object Model.
- ODL should not be a full programming language, but rather a specification language for interface signatures.
- ODL should be programming-language independent.
- ODL should be compatible with the OMG's Interface Definition Language (IDL).
- ODL should be extensible, not only for future functionality, but also for physical optimizations.
- ODL should be practical, providing value to application developers, while being supportable by the ODBMS vendors within a relatively short time frame after publication of the specification.

ODL is not intended to be a full programming language. It is a specification language for interface signatures. Database management systems (DBMSs) have traditionally provided interfaces that support data definition (using a Data Definition Language — DDL) and data manipulation (using a Data Manipulation Language — DML). The DDL allows users to define their data types and interfaces. DML allows programs to create, delete, read, change, etc., instances of those data types. The ODL described in this chapter is a DDL for object types. It defines the characteristics of types, including their properties and operations. ODL defines only the signatures of operations and does not address definition of the methods that implement those operations. ODMG-93 does

not provide a standard OML. Chapters 5 and 6 define standard APIs to bind conformant ODBMSs to C++ and to Smalltalk.

The ODL is intended to define object types that can be implemented in a variety of programming languages. Therefore ODL is not tied to the syntax of a particular programming language; users can use ODL to define schema semantics in a programming-language independent way. A schema specified in ODL can be supported by any ODMG-compliant ODBMS and by mixed-language implementations. This portability is necessary for an application to be able to run with minimal modification on a variety of ODBMSs. Some applications may in fact need simultaneous support from multiple ODBMSs. Others may need to access objects created and stored using different programming languages. ODL provides a degree of insulation for applications against the variations in both programming languages and underlying ODBMS products.

The C++ ODL and Smalltalk ODL bindings defined in Chapters 5 and 6 respectively are designed to fit smoothly into the declarative syntax of their host programming language. Due to the differences inherent in the object models native to these programming languages, it is not always possible to achieve consistent semantics across the programming-language specific versions of ODL. Our goal has been to minimize these inconsistencies, and we have noted in Chapters 5 and 6 the restrictions applicable to each particular language binding.

The syntax of ODL extends IDL —the Interface Definition Language developed by the OMG as part of the Common Object Request Broker Architecture (CORBA). IDL was itself influenced by C++, giving ODL a C++ flavor. Appendix B, “ODBMS in the OMG ORB Environment,” describes the relationship between ODL and IDL. ODL adds to IDL the constructs required to specify the complete semantics of the ODMG Object Model.

ODL also provides a context for integrating schemas from multiple sources and applications. These source schemas may have been defined with any number of object models and data definition languages; ODL is a sort of lingua franca for integration. For example, various standards organizations like STEP/PDES (EXPRESS), ANSI X3H2 (SQL), ANSI X3H7 (Object Information Management), CFI (CAD Framework Initiative), and others have developed a variety of object models and, in some cases, data definition languages. Any of these models can be translated to an ODL specification (Figure 3-1). This common basis then allows the various models to be integrated with common semantics. An ODL specification can be realized concretely in an object programming language like C++ or Smalltalk.

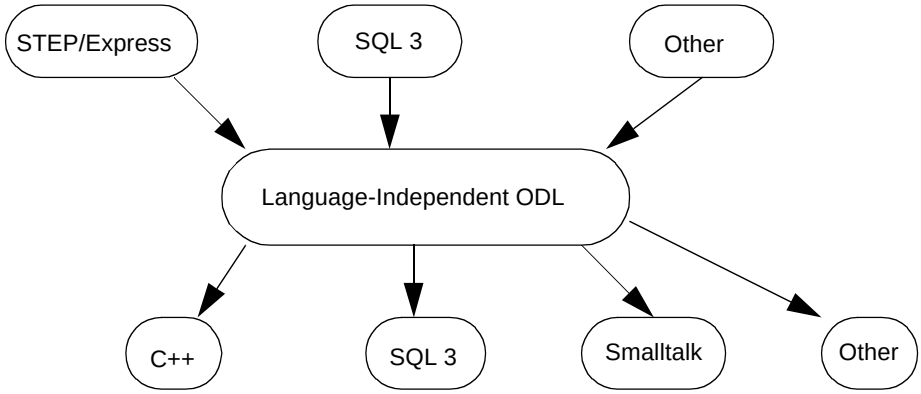


Figure 3-1. ODL Mapping to Other Languages

3.2 Specification

A type is defined by specifying its interface in ODL. The top-level BNF for ODL is as follows:

```
<interface_dcl>      ::= <interface_header>
                         [ : <persistence_dcl> ] { [ <interface_body> ] } ;
<persistence_dcl>    ::= persistent | transient
<interface_header>   ::= interface <identifier>
                         [ <inheritance_spec> ]
                         [ <type_property_list> ]
```

The characteristics of the type itself appear first, followed by lists that define the properties and operations of its interface. Any list may be omitted if it is not applicable in the interface.

3.2.1 Type Characteristics

Supertype information, extent naming, and specification of keys (i.e., uniqueness constraints) are all characteristics of types, but do not apply directly to the types' instances. The BNF for type characteristics follows.

```
<inheritance_spec>   ::=  : <scoped_name> [ , <inheritance_spec> ]
<type_property_list>
                     ::=  ( [ <extent_spec> ] [ <key_spec> ] )
<extent_spec>        ::=  extent <string>
<key_spec>           ::=  key[s] <key_list>
<key_list>           ::=  <key> | <key> , <key_list>
<key>                ::=  <property_name> | ( <property_list> )
<property_list>      ::=  <property_name>
                           | <property_name> , <property_list>
<property_name>      ::=  <identifier>
<scoped_name>        ::=  <identifier>
                           | :: <identifier>
                           | <scoped_name> :: <identifier>
```

Each supertype must be specified in its own type definition. Each attribute or relationship traversal path named as (part of) a type's key must be specified in the key_spec of the type definition. The extent and key definitions may be omitted if inapplicable to the type being defined. A type definition should include no more than one extent or key definition.

A simple example for the interface definition of a Professor type is

```
interface Professor: Person
(       extent professors
        keys faculty_id, soc_sec_no): persistent
{
        properties
        operations
};
```

Keywords are highlighted.

3.2.2 Instance Properties

A type's instance properties are the attributes and relationships of its instances. These properties are specified in attribute and relationship specifications. The BNF follows.

```
<interface_body>     ::=  <export> | <export> <interface_body>
<export>             ::=   <type_dcl> ;
                           | <const_dcl> ;
                           | <except_dcl> ;
                           | <attr_dcl> ;
                           | <rel_dcl> ;
                           | <op_dcl> ;
```

3.2.2.1 Attributes

The BNF for specifying an attribute follows.

```
<attr_dcl>          ::=  [ readonly ] attribute
                         <domain_type> <attribute_name>
                          [ <fixed_array_size>]
<domain_type>       ::=  <simple_type_spec>
                         | <struct_type>
                         | <enum_type>
```

For example, adding attribute definitions to the Professor type's ODL specification:

```
interface Professor: Person
(           extent professors
            keys faculty_id, soc_sec_no): persistent
{
            attribute string name;
            attribute unsigned short faculty_id[6];
            attribute long soc_sec_no[10] ;
            attribute Address address;
            attribute set<string> degrees;
            relationships
            operations
};
```

Note that the keyword attribute is mandatory.

3.2.2.2 Relationships

A relationship specification names and defines a traversal path for a relationship. A traversal path definition includes designation of the target type, ordering information, and information about the inverse traversal path found in the target type. The BNF for relationship specification follows.

```
<rel_dcl>                ::=  relationship
                              <target_of_path> <identifier>
                               inverse <inverse_traversal_path>
                              [ { order_by <attribute_list> } ]
<target_of_path>         ::=  <identifier>
                              | <rel_collection_type> < <identifier> >
<inverse_traversal_path>
                         ::=  <identifier> :: <identifier>
<attribute_list>         ::=   <scoped_name>
                               | <scoped_name> , <attribute_list>
```

Traversal path cardinality information is included in the specification of the target of a traversal path. The target type must be specified with its own type definition, unless the relationship is recursive. Use of the collection_type option of the BNF indicates cardinality greater than one on the target side. If this option is omitted, the cardinality on the target side is one. The most commonly used collection types are expected to be Set, for unordered members on the target side of a traversal path, and List, for ordered members on the target side. Bags are supported as well. An ordering criterion is specified with the order_by clause. Each attribute used in an ordering criterion must be defined in the attribute list of the target type's definition. The inverse traversal path must be defined in the property list of the target type's definition. If an inverse traversal path is not specified, the relationship is considered to be unidirectional. For example, adding relationships to the Professor type's interface specification:

```
interface Professor: Person
(       extent professors
        keys faculty_id, soc_sec_no): persistent
{
        attribute string name;
        attribute unsigned short faculty_id[6];
        attribute long soc_sec_no[10] ;
        attribute Address address;
        attribute set<string> degrees;
        relationship set<Student> advises inverse Student::advisor;
        relationship set<TA> teaching_assistants inverse TA::works_for;
        relationship Department department
                inverse Department::faculty;
        operations
};
```

The keyword relationship is mandatory.

Note that the attribute and relationship specifications can be mixed in the property list. It is not necessary to define all of one kind of property, then all of the other kind.

3.2.3 Operations

ODL is compatible with IDL for specification of operations. The high-level BNF for operations follows.

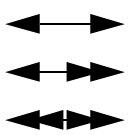
```
<op_dcl>            ::=  [ oneway ] <op_type_spec> <identifier>
                         <parameter_dcls> [<raises_expr>]
                         [<context_expr>]
<op_type_spec>      ::=  <simple_type_spec>
                         | void
<parameter_dcls>    ::=  ( [ <param_dcl_list> ] )
<param_dcl_list>    ::=  <param_dcl>
                         | <param_dcl> , <param_dcl_list>
```

```
<param_dcl>          ::= <param_attribute><simple_type_spec><declarator>
<param_attribute>    ::=  in | out| inout
<raises_expr>        ::=  raises ( <scoped_name_list>)
<context_expr>       ::=  context ( <string_literal_list> )
<scoped_name_list>   ::=  <scoped_name>
                          | <scoped_name> , <scoped_name_list>
<string_literal_list>::=  <string_literal>
                          | <string_literal> , <string_literal_list>
```

See Section 3.5 for the full BNF for operation specification.

3.3 An Example in ODL

This section illustrates the use of ODL to declare the schema for a sample application based on a university database. The object types in the sample application are shown as rectangles in Figure 3-2. Relationship types are shown as lines. The cardinality permitted by the relationship type is indicated by the arrows on the ends of the lines:



one-to-one

one-to-many

many-to-many

In the example, the type Professor is a subtype of the type Employee, and the type TA (for Teaching Assistant) is a subtype of both Employee and Student. Large gray arrows run from subtype to supertype in the figure.

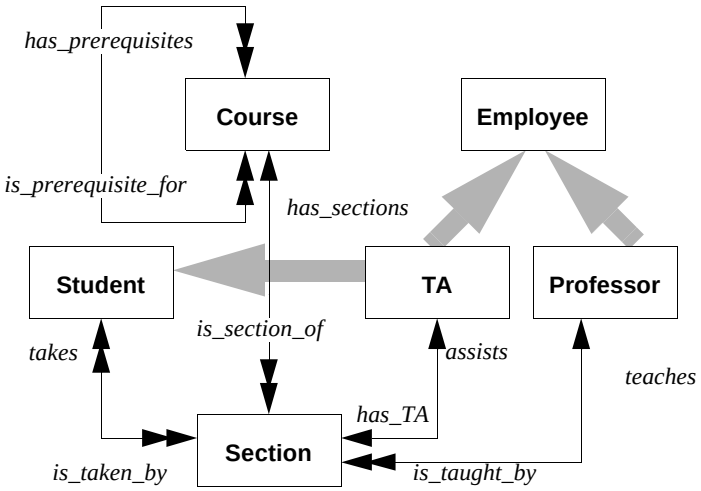


Figure 3-2. Graphical Representation of Schema

An ODL definition for the interfaces of the schema's types follows:

```
interface Course
(       extent courses
        keys name, number)
{
        attribute string name;
        attribute string number;
        relationship list<Section> has_sections
            inverse Section::is_section_of
            {order_by Section::number};
        relationship set<Course> has_prerequisites
             inverse Course::is_prerequisite_for;
        relationship set<Course> is_prerequisite_for
             inverse Course::has_prerequisites;

        boolean offer (in unsigned short semester) raises (already_offered);
        boolean drop (in unsigned short semester) raises (not_offered);
};
interface Section
(       extent sections
        key (is_section_of, number))
{
        attribute string number;
        relationship Professor is_taught_by inverse Professor::teaches;
        relationship TA has_TA inverse TA::assists;
        relationship Course is_section_of inverse Course::has_sections;
        relationship set<Student> is_taken_by inverse Student::takes;
};

interface Employee
(       extent employees
        key (name, id))
{
        attribute string name;
        attribute short id;
        attribute unsigned short annual_salary;
        void hire ();
        void fire () raises (no_such_employee);
};
```

```
interface Professor: Employee
(       extent professors)
{
        attribute enum Rank {full, associate, assistant} rank;
        relationship set<Section> teaches inverse Section::is_taught_by;
        short grant_tenure () raises (ineligible_for_tenure);
};

interface TA: Employee, Student
()
{
        relationship Section assists inverse Section::has_TA;
};

interface Student
(       extent students
        keys name, student_id)
{
        attribute string name;
        attribute string student_id;
        attribute struct Address {string college, string room_number}
            dorm_address;
        relationship set<Section> takes inverse Section::is_taken_by;
        boolean register_for_course (in unsigned short course,
            in unsigned short Section)
            raises (unsatisfied_prerequisites, section_full, course_full);
        void drop_course (in unsigned short Course)
            raises (not_registered_for_that_course);
        void assign_major (in unsigned short Department);
        short transfer (in unsigned short old_section,
            in unsigned short new_section)
            raises (section_full, not_registered_in_section);
};
```

3.4 Another Example

Following is another example that will be used as an illustration of ODL. The same example will be used in Chapter 5 to illustrate the binding of ODL to C++. The application manages personnel records. The database manages information about people, their marriages, children, and history of residences. Person has an extent named people. A Person has name, address, spouse, children, and parents properties. The operations birth, marriage, ancestors, and move are also characteristics of Person: birth

adds a new child to the children list of a Person instance, marriage defines a spouse for a Person instance, ancestors computes the set of Person instances who are the ancestors of a particular Person instance, and move changes a Person instance's address. An Address is a structure whose properties are number, street, and city_name; number is of type unsigned short, street and city are of type string. City has properties city_code, name, and population. City_code is of type unsigned short; name is of type string; population is a set of Refs to Person objects. Spouse is a traversal path to a spouse:spouse 1:1 recursive relationship; children is one of the traversal paths of a children:parents m:n recursive relationship. The children of a given Person instance are ordered by birth_date. Parents is the other traversal path of the children:parents relationship.

The ODL specifying the interfaces for this schema follows.

```
interface Person
(       extent people)
{
        attribute string name;
        attribute struct Address {unsigned short number, string street,
            string city_name} address;
        relationship Person spouse inverse Person::spouse;
        relationship set<Person> children inverse Person::parents
            {order_by birth_date} ;
        relationship list<Person> parents inverse Person::children;
        void birth (in string name);
        boolean marriage (in string person_name) raises (no_such_person);
        unsigned short ancestors (out set<Person> all_ancestors)
            raises (no_such_person);
        void move (in string new_address);
};

interface City
(       extent cities
        key city_code)
{
        attribute unsigned short city_code;
        attribute string name;
        attribute set<Person> population;
};
```

3.5 ODL Grammar

Following is the complete BNF for the ODL, which includes the IDL. The numbers on the production rules match their numbers in the OMG CORBA specification. Modified production rules have numbers suffixed by an asterisk, e.g., (5*). New production rules have alpha extensions, e.g., (5a).

(1) <specification> ::= <definition>
| <definition> <specification>

(2) <definition>::= <type_dcl> **;**
| <const_dcl> **;**
| <except_dcl> **;**
| <interface> **;**
| <module> **;**

(3) <module>::= **module** <identifier> **{** <specification> **}**

(4) <interface>::= <interface_dcl>
| <forward_dcl>

(5*) <interface_dcl>::= <interface_header>
[**:** <persistence_dcl>] **{** [<interface_body>] **}**

(5a) <persistence_dcl>::= **persistent** | **transient**

(6) <forward_dcl>::= **interface** <identifier>

(7*) <interface_header>::= **interface** <identifier>
[<inheritance_spec>]
[<type_property_list>]

(7a) <type_property_list>
::= **(** [<extent_spec>] [<key_spec>] **)**

(7b) <extent_spec>::= **extent** <string>

(7c) <key_spec>::= **key**[**s**] <key_list>

(7d) <key_list>::= <key> | <key> **,** <key_list>

(7e) <key>::= <property_name> | (<property_list>)

(7f) <property_list>::= <property_name>
| <property_name> **,** <property_list>

(7g) <property_name>::= <identifier>

(8) <interface_body>::=
<export> | <export> <interface_body>

(9*) <export>::= <type_dcl>**;**
| <const_dcl>**;**
| <except_dcl>**;**
| <attr_dcl>**;**
| <rel_dcl>**;**
| <op_dcl>**;**

(10) <inheritance_spec>::=
: <scoped_name> [, <inheritance_spec>]

(11) <scoped_name>::= <identifier>
| **::** <identifier>
| <scoped_name> **::** <identifier>

(12) <const_dcl>::= **const** <const_type> <identifier> **=**
<const_exp>

(13) <const_type>::= <integer_type>
| <char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <scoped_name>

(14) <const_exp>::= <or_expr>

(15) <or_expr>::= <xor_expr>
| <or_expr> | <xor_expr>

(16) <xor_expr>::= <and_expr>
| <xor_expr> **^** <and_expr>

(17) <and_expr>::= <shift_expr>
| <and_expr> **&** <shift_expr>

(18) <shift_expr>::= <add_expr>
| <shift_expr> **>>** <add_exp>
| <shift_expr> **<<** <add_expr>

(19) <add_expr>::= <mult_expr>
| <add_expr> **+** <mult_expr>
| <add_expr> **-** <mult_expr>

(20) <mult_expr>::= <unary_expr>
| <mult_expr> * <unary_expr>
| <mult_expr> **/** <unary_expr>
| <mult_expr> **%** <unary_expr>

(21) <unary_expr>::= <unary_operator> <primary_expr>
| <primary_expr>

(22) <unary_operator>::= **-**
| **+**
| **~**

(23) <primary_expr>::= <scoped_name>
| <literal>
| **(** <const_exp> **)**

(24) <literal>::= <integer_literal>
| <string_literal>
| <character_literal>
| <floating_pt_literal>
| <boolean_literal>

(25) <boolean_literal>::= **TRUE**
| **FALSE**

(26) <positive_int_const>::= <const_exp>
(27) <type_dcl>::= **typedef** <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>
(28) <type_declarator>::= <type_spec> <declarators>
(29) <type_spec>::= <simple_type_spec>
| <constr_type_spec>
(30) <simple_type_spec>::= <base_type_spec>
| <template_type_spec>
| <scoped_name>
(31) <base_type_spec>::= <floating_pt_type>
| <integer_type>
| <char_type>
| <boolean_type>
| <octet_type>
| <any_type>
(32*) <template_type_spec>::= <array_type>
| <string_type>
| <coll_type>
(32a) <coll_type> ::= <coll_spec> **<** <simple_type_spec> **>**
(32b) <coll_spec> ::= **set** | **list** | **bag**
(33) <constr_type_spec>::= <struct_type>
| <union_type>
| <enum_type>
(34) <declarators>::= <declarator>
| <declarator> **,** <declarators>
(35) <declarator>::= <simple_declarator>
| <complex_declarator>
(36) <simple_declarator>::= <identifier>
(37) <complex_declarator>::= <array_declarator>
(38) <floating_pt_type>::= **float**
| **double**
(39) <integer_type>::= <signed_int>
| <unsigned_int>
(40) <signed_int>::= <signed_long_int>
| <signed_short_int>
(41) <signed_long_int>::= **long**
(42) <signed_short_int>::= **short**
(43) <unsigned_int>::= <unsigned_long_int>
| <unsigned_short_int>
(44) <unsigned_long_int>::= **unsigned long**
(45) <unsigned_short_int>::= **unsigned short**

(46) <char_type>::= **char**
(47) <boolean_type>::= **boolean**
(48) <octet_type>::= **octet**
(49) <any_type>::= **any**
(50) <struct_type>::= **struct** <identifier> **{** <member_list> **}**
(51) <member_list>::= <member> | <member>
<member_list>
(52) <member>::= <type_spec> <declarators> **;**
(53) <union_type>::= **union** <identifier> **switch**
(<switch_type_spec> **) {** <switch_body> **}**
(54) <switch_type_spec>::= <integer_type>
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>
(55) <switch_body>::= <case> | <case> <switch_body>
(56) <case>::= <case_label_list> <element_spec> **;**
(56a) <case_label_list>::= <case_label>
| <case_label> <case_label_list>
(57) <case_label>::= **case** <const_exp> **:**
| **default :**
(58) <element_spec>::= <type_spec> <declarator>
(59) <enum_type>::= **enum** <identifier> **{** <enumerator_list> **}**
(59a) <enumerator_list>::= <enumerator>
| <enumerator> **,** <enumerator_list>
(60) <enumerator>::= <identifier>
(61*) <array_type>::= <array_spec> **<** <simple_type_spec> **,**
<positive_int_const> **>**
| <array_spec> **<** <simple_type_spec> **>**
(61a*) <array_spec>::= **array** | **sequence**
(62) <string_type>::=**string <** <positive_int_const> **>**
| **string**
(63) <array_declarator>::= <identifier> <array_size_list>
(63a) <array_size_list>::= <fixed_array_size>
| <fixed_array_size> <array_size_list>
(64) <fixed_array_size>::= **[** <positive_int_const> **]**
(65*) <attr_dcl> ::= [**readonly**] **attribute**
<domain_type> <attribute_name>
[<fixed_array_size>]
(65a) <domain_type>::= <simple_type_spec>
| <struct_type>
| <enum_type>

(65b) <rel_dcl> ::= **relationship**
<target_of_path>
<identifier>
inverse <inverse_traversal_path>

(65c) <target_of_path>::= <identifier>
| <rel_collection_type> **<** <identifier> **>**

(65d) <inverse_traversal_path>::=
<identifier> **::** <identifier>

(65e) <attribute_list>::= <scoped_name>
| <scoped_name> **,** <attribute_list>

(65f) <rel_collection_type::= **set** | **list** | **bag** | **array**

(66) <except_dcl>::= **exception** <identifier>
{ [<member_list>] **}**

(67) op_dcl>::= [<op_attribute>] <op_type_spec>
<identifier> <parameter_dcls>
[<raises_expr>] [<context_expr>]

(68) <op_attribute>::= **oneway**

(69) <op_type_spec>::= <simple_type_spec>
| **void**

(70) <parameter_dcls>::= **(** [<param_dcl_list>] **)**

(70a) <param_dcl_list>::= <param_dcl>
| <param_dcl> **,** <param_dcl_list>

(71) <param_dcl>::= <param_attribute> <simple_type_spec>
<declarator>

(72) <param_attribute>::= **in**
| **out**
| **inout**

(73) <raises_expr>::= **raises (** <scoped_name_list> **)**

(73a) <scoped_name_list>::= <scoped_name>
| <scoped_name> **,**
<scoped_name_list>

(74) <context_expr>::= **context (** <string_literal_list> **)**

(74a) <string_literal_list>::= <string_literal>
| <string_literal> **,** <string_literal_list>