Lenguajes de ODMG-93

Amparo López Gaona

17 Septiembre 1997

Resumen

En este trabajo se presentan el lenguaje para definición de objetos, el lenguaje para consulta y la forma en que se puede integrar ODMG-93 con lenguajes orientados a objetos como C++.

1 Introducción

ODMG-93 es la integración de un modelo de objetos (OMG), lenguaje para la consulta de bases de datos (SQL) y lenguajes orientados a objetos (C++, Smalltalk, Java) con objeto de definir un estándar para desarrollar aplicaciones usando sistemas gestores de bases de datos orientadas a objetos. En el artículo anterior se describe el modelo de objetos sobre el cual trabaja ODMG-93, en este trabajo se detallan el lenguaje para definición de objetos, el lenguaje para consulta y la forma en que se puede integrar ODMG-93 con lenguajes orientados a objetos como C++.

2 Lenguaje de definición de objetos

ODL (*Object Definition Language*) tiene como objetivos minimizar las inconsistencias entre el modelo de objetos y los lenguajes de programación, facilitar la portabilidad entre esquemas de bases de datos cuyos sistemas manejadores de bases de datos (SGBD) sigan el estándar ODMG-93 y facilitar la interoperabilidad entre distintos SGBDOs [1], a diferencia de CORBA

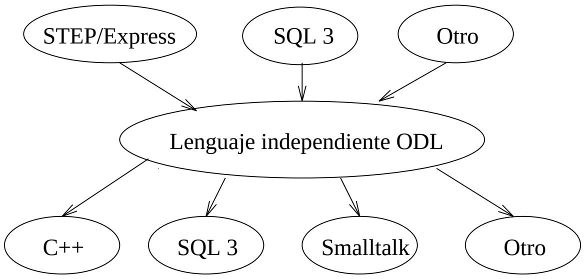

Figura 1: Mapeo de ODL a C++.

que permite que una aplicación que corre sobre un sistema `X` conforme al modelo CORBA lo haga sobre un sistema `Y` que también es conforme a CORBA [3]. ODL proporciona un contexto para integrar esquemas definidos para diversos modelos de objetos y lenguajes de descripción de datos. Cualquiera de ellos puede traducirse a una especificación ODL y ésta concretarse en algún lenguaje de programación orientado a objetos (Ver la figura 1).

Para facilitar la presentación de ODL se definen las interfaces de un esquema en el que se tienen personas que pueden ser empleados o estudiantes. Toda persona tiene nombre, fecha de nacimiento, un domicilio, padres e hijos. Los empleados tienen además un salario y los estudiantes un grado de estudios. El domicilio es un apartamento que pertenece a un edificio ubicado en alguna calle con número.

En la definición de una interfaz aparecen primero las características del tipo (tales como extensión y claves) seguidas, opcionalmente, por listas de propiedades y operaciones. Por ejemplo la definición de la intefaz para el tipo `Persona` se vería así:

interface Persona (extent gente key dni) : persistent
{
    definicion de atributos
    definicion de relaciones
    definicion de operaciones
};

En esta definición se está especificando que la extensión del tipo `Persona`, se llamará `gente,` que como clave tiene el `dni` (documento nacional de identidad) y que sus instancias serán **persistent**es. La extensión y la clave pueden omitirse, en caso de incluirse no debe aparecer más de una vez la palabra `extent, key` o `keys`.

Los atributos se definen empezando con la palabra `attribute` seguida por el tipo del atributo, que puede ser uno de los predefinidos en el modelo de objetos o definido por el usuario, y finalmente el nombre del atributo. Esto se muestra en el siguiente ejemplo.

```
interface Persona (extent gente  key dni) : persistent
{
     attribute String nombre;
     attribute Date   fecha_nac;
};
```

Para cada atributo se generan los métodos (`get_value_` y `set_value_`) para acceder a cada atributo. Para el ejemplo que se muestra se generarían los métodos `get_value_nombre, get_value_fecha_nac, set_value_nombre` y `set_value_fecha_nac`

La especificación de una relación da nombre y define un enlace bidireccional es decir, si un objeto x tiene una relación r con un objeto y, entonces hay una relación inversa de y a x. La definición de la relación incluye el tipo destino, el orden de la información y la información sobre el enlace inverso de este destino. Si en la especificación del destino se escribe alguno de los tipos colección (`bag, set` o `list`) se asume que la funcionalidad del enlace es N, en caso contrario es uno.

```
interface Persona (extent gente  key dni) : persistent
{
     attribute String nombre;
     attribute Date   fecha_nac;

     relationship  Set<Persona> padres inverse Persona::hijos;
     relationship  List<Persona> hijos inverse Persona::padres
```

```
        {order_by fecha_nac};
    relationship  Apartamento vive_en inverse Apartamento::es_usado_por;
};
```

En el ejemplo se definieron tres relaciones: `padres` de funcionalidad N con el tipo `Persona` y cuya inversa es `hijos`. La otra relación también es con el tipo `Persona` y define la relación `hijos` cuya inversa es `padres` se pide que la relación esté ordenada por la fecha de nacimiento de cada hijo. La tercera relación es de funcionalidad uno con el tipo `Apartamento` llamada `vive_en`, y cuya inversa es `es_usado_por`.

Para definir relaciones es obligatoria la palabra `relationship`, en tanto que la palabra `inverse` es opcional y sirve para asegurar las restricciones de integridad referencial, por ejemplo si una persona se muda de apartamento, el atributo `es_usado_por` se pone automáticamente en `NULL` hasta que otra persona tome este apartamento. De igual manera, si un apartamento se elimina, el atributo `vive_en` se pone automáticamente en `NULL` para evitar referencias dañadas.

Con respecto a las operaciones cabe mencionar que la sintaxis de las signaturas es similar a la de C++, excepto que cada parámetro va precedido de la palabra `in`, `out` o `inout` según sea el uso que se le dará al mismo.

```
interface Persona ( extent gente  key dni) : persistent
{
    attribute String nombre;
    attribute Date   fecha_nac;

    relationship  Set<Persona> padres inverse Persona::hijos;
    relationship  List<Persona> hijos inverse Persona::padres
        {order_by fecha_nac};
    relationship  Apartamento vive_en inverse apartamento::es_usado_por;

    int edad();
    void matrimonio(in Persona);
    Boolean casado (in string nombre) raises(no_existe_persona);
    void nacimiento(in Persona);
```

```
    Set<Persona> ancestros() raises(no_existe_persona);
};
```

El tipo `Empleado` se define como subtipo de `Persona`, agregando a las características de `Persona` un salario y un conjunto de actividades.

```
interface Empleado: Persona
{
   attribute float salario;
   set<string> actividades();
};
```

Un `Apartamento` tiene número, pertenece a un edificio y es usado por una persona. Con el fin de ilustrar una relación 1:1 se asume que en todo apartamento vive sólo una persona.

```
interface Apartamento (extent apartamentos);
{
   attribute numero;
   relationship Edificio construido_en inverse Edificio::contiene;
   relationship Persona es_usado_por inverse Persona::vive_en;
};
```

Por cuestiones de espacio se omite la especificación de la interfaz para `estudiantes`, `edificio` y `direccion`.

3 Lenguaje de consulta

El lenguaje de consulta OQL proporciona facilidades para especificar consultas sobre esquemas de bases de datos definidos usando ODL. Un programa OQL consta de un conjunto, posiblemente vacío, de definiciones de consulta seguidos de una expresión. Tal conjunto es no recursivo aunque una consulta pueda llamar una operación que invoque recursivamente a una consulta. El resultado de una definición de consulta, es una consulta-con-nombre la cual es válida sólo dentro del ámbito del programa OQL [2]. La forma general de una expresión de consulta es:

select *constructor(p1, p2,...,pn)*
from *lista de extensiones*
where *condicion*

El *constructor* determina el tipo del resultado. Para cada combinación de objetos en la lista de extensiones se evalúa la condición; si la combinación de objetos satisface la condición, se aplica el constructor y su resultado es regresado. La *condición* es una expresión Booleana construida a partir de referencias a atributos y llamadas a métodos. Es fuertemente tipado, y permite consultas sobre objetos que pueden regresar objetos y literales. Es posible crear un objeto con OID usando un constructor de tipo que contenga las propiedades del objeto y opcionalmente valores iniciales para ellas, todo encerrado en paréntesis. Las expresiones de consulta definen operaciones cuyo resultado tiene tipo que puede inferirse de la expresión en sí.

Para navegar dentro de la base de datos a partir de un objeto se utiliza la notación punto "." con la cual se puede entrar a objetos complejos así como seguir relaciones. Con el esquema descrito en la sección anterior, la consulta en OQL para obtener el nombre de la calle donde vive una Persona `p` se escribiría de la siguiente manera:

```
p.vive_en.construido_en.direccion.calle
```

esta consulta empieza en un objeto de tipo `Persona`, de ahí llega a un `Apartamento` (por la relación `vive_en`), luego a un `Edificio` (`construido_en`) y sigue dentro del atributo de tipo `Dirección` para finalmente conseguir el nombre de la calle.

Puede suponerse que la consulta en OQL para conocer el nombre de cada hijo de una persona `p`, es `p.hijos.nombre`. Esto es incorrecto debido a que `hijos` es una lista de referencias, se debe usar una instrucción `select_from_where`, como si se tratara de SQL. La consulta correcta es

```
select h.nombre
from h in p.hijos
```

Para seleccionar datos que satisfagan condiciones se usa la cláusula `where`. Por ejemplo, si se desea obtener la dirección de las personas que no viven con sus padres, tienen al menos un hermano y su padre vive en la Gran Vía se escribe así:

```
select h.vive_en.construido_en.direccion
from p in Personas,
     h in p.hijos
where p.vive_en.construido_en.direccion.calle = "Gran Via" and
    count(p.hijos) >= 2 and h.vive_en != p.vive_en
```

OQL puede crear cualquier valor complejo como resultado final, o dentro de una consulta como cálculo intermedio. Por ejemplo, para obtener la dirección de los hijos de cada persona junto con la dirección de la persona en cuestión puede hacerse la siguiente consulta:

```
select struct(yo:p.nombre,
   mi_direccion:p.vive_en.construido_en.direccion,
   mis_hijos:(select struct(nombre:h.nombre,
                     direccion:h.vive_en.construido_en.direccion)
              from h in p.hijos))
from p in Personas
```

con ello para cada persona se obtiene, en una estructura, su nombre, dirección, así como el nombre y la dirección de cada hijo.

Para crear objetos complejos OQL usa el nombre de una clase como constructor, los atributos del objeto pueden inicializarse explícitamente por cualquier expresión válida. Por ejemplo, para crear un nuevo edificio con dos apartamentos, suponiendo que se tiene definido un tipo llamado `Lista_apart`, se hará la siguiente consulta:

```
Edificio(direccion:struct(numero:10, calle:"Gran Via"),
  apartamentos:Lista_apart(Apartamento(numero:1),
                           Apartamento(numero:2)))
```

Los parámetros en el paréntesis permiten inicializar propiedades del objeto.

Desde OQL se pueden llamar métodos en cualquier sitio donde el tipo resultante concuerde con el esperado en la consulta. La notación es la misma que para acceder a un atributo o recorrer una relación. Por ejemplo, para calcular la edad del primogénito de "Juan" se podría hacer:

```
select max(select h.edad from h en p.hijos)
from p in Personas,
where p.nombre = "Juan"
```

Desde luego, un método puede regresar algo más complejo que esto por ejemplo, dentro de un edificio `e` se desea saber quiénes habitan el apartamento más barato.

```
e.menos_caro.es_usado_por.nombre
```

aunque `menos_caro` es un método, se recorre como si fuera una relación.

OQL es un lenguaje funcional y por tanto todos los operadores pueden componerse libremente. Por ejemplo, si se desea saber el nombre de la calle donde viven los empleados con el menor salario promedio que en otras calles, se puede escribir:

```
(sort s in (
   group e in (select (Empleado)p from p in Personas
         where "tiene un trabajo" in p.actividades)
   by (calle: e.vive_en.construido_en.direccion.calle)
   with (salario_promedio:avg(select x.salario
                              from x in partition)))
 by s.salario_promedio)[0].calle
```

Como lenguaje embebido OQL permite consultar objetos soportados por un lenguaje nativo mediante expresiones que llevan átomos, estructuras, colecciones y literales. Una consulta es una función que al aplicarse a una entrada, devuelve un objeto cuyo tipo puede inferirse del operador que forma la expresión de la consulta. En la siguiente sección se muestra un ejemplo de ello.

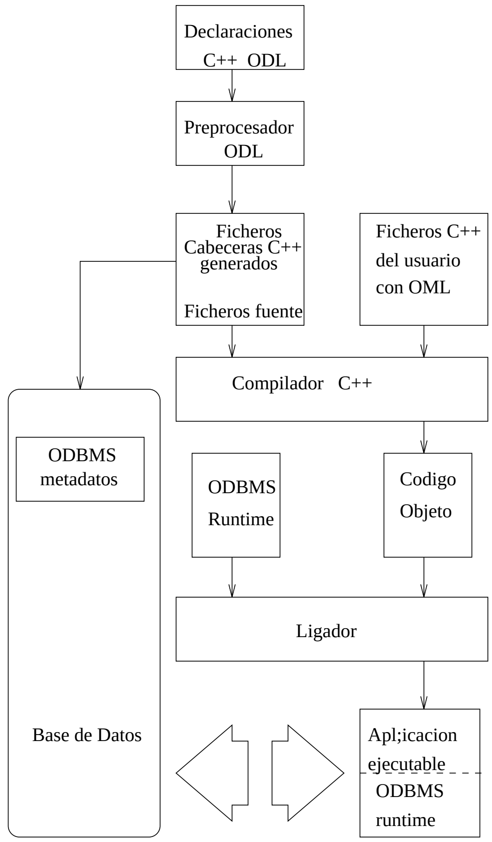

Figura 2: Mapeo de ODL a otros lenguajes.

4 Enlace con lenguajes de programación

ODMG-93 define enlaces con C++ y con Smalltalk, con un mapeo directo de los conceptos del MO de ODMG a los conceptos del MO del lenguaje de programación.

La figura 2 muestra el uso de los componentes del enlace con C++. La especificación describe posibles extensiones de este proceso de enlace que incluye un preprocesador capaz de procesar cuerpos de métodos, así como declaraciones de clases.

El enlace de C++ hace el mapeo del MO de ODMG en C++ mediante un conjunto de clases llamadas "clases capaces de persistencia" cuyas instancias pueden ser persistentes o transientes, para ello se define una superclase denominada `Persistent_Object`. Para cada clase `T` capaz de persistencia, se define otra clase `Ref<T>` de tal manera que las instancias de T pueden referenciarse usando estas clases parametrizadas.

Como ejemplo de la definición de una clase usando el enlace con C++ se tiene:

```
class Persona : public Persitent_Object {
public:
// atributos
     String nombre;
     Date   fecha_nac;

// relaciones
     Set<Ref<Persona>> padres inverse Persona::hijos;
     List<Ref<Persona>> hijos inverse Persona::padres;
         {order_by fecha_nac};
     Ref<Apartamento> vive_en inverse apartamento::es_usado_por;

// operaciones:
     int edad();
     void matrimonio(Persona);
     Boolean casado (string) raises(no_existe_persona);
     void nacimiento(Persona);
     Set<Ref<Persona>> ancestros;

// extension:
     Ref<Set<Ref<Persona>>>gente;
     static const char * const nombre_extension;

private:
   ...
};
```

La sintaxis para declaración de clases en C++ ODL es idéntica a la declaración de clases para C++, excepto que incluye la declaración de relaciones que se distinguen de los miembros por la presencia de la palabra `inverse`. En el ejemplo se muestra que la definición de una clase tiene dos partes: la interfaz (parte pública) y la implementación (miembros protegidos y privados así como definición de funciones). Se debe recordar que en ODMG está permitido tener más de una implementación para una clase dada.

El principio básico de diseño del enlace con C++ es que la sintaxis utilizada para trabajar con objetos persistentes y la que se usa para trabajar con objetos transientes deberían ser idénticas hasta donde sea posible, aunque de momento si existe diferencia, por ejemplo, las consultas sólo pueden hacerse sobre objetos persistentes. Las instancias de clases capaces de persistencia pueden contener miembros de los tipos primitivos de C++, clases definidas por el usuario, o apuntadores a datos transientes. Las aplicaciones pueden hacer referencia a estos miembros usando apuntadores (*) o referencias (&) de C++ sólo durante la ejecución de la transacción. Cuando la transacción termina bien (comitted) el valor de ellos se hace 0.

La creación de objetos es mediante el operador `new`, el cual está sobrecargado para aceptar argumentos adicionales que especifican el tiempo de vida de un objeto. El acceso a objetos persistentes se hace mediante métodos definidos sobre las clases: `Ref, Persistent_Object` y `Database`. El operador `->` se usa para acceder a miembros de objetos persistentes referenciados, tal operación garantiza que si el objeto referenciado existe será devuelto y en caso contrario se produce un error.

Las transacciones pueden iniciarse, terminarse bien, abortarse y marcarse (checkpointed). Es importante notar que todos los accesos, creaciones, modificaciones y supresiones de objetos persistentes deben hacerse dentro de una transacción. Las transacciones se implementan en C++ como objetos de la clase `Transaction`. Los objetos del tipo `Database`, al igual que las transacciones, son transientes. Las bases de datos deben abrirse antes de iniciar cualquier transacción que las use y deben cerrarse al terminar estas transacciones.

A continuación se muestra un programa para crear un objeto persistente del tipo `Apartamento:`

```
Transaction mudanza;
mudanza.begin();
   Ref<Apartamento>casa = new(database)Apartamento;
   Ref<Persona>juan = database->lookup_object("Juan");
                // Localiza a Juan
   Apartamento.insert_element(casa);  // Coloca el nuevo apartamento
   juan.vive_en = casa;
mudanza.commit();
```

una vez iniciada la transacción `mudanza` se crea un nuevo apartamento en la base de datos y su referencia se llama `casa`. Luego se busca un objeto denominado "Juan", se inserta el apartamento en la extensión de `Apartamento` y se estable la conexión de que Juan vive ahí. Finalmente se hace commit.

El enlace con C++ también incluye un mapeo de OQL a la semántica de C++, las consultas pueden especificarse desde cualquier programa en C++ usando un método `query` definido para todas las clases `Collection`, o usando una función `oql` que no está asociada con ninguna clase. El método `query` toma una cadena que representa una consulta y ésta se evalúa en tiempo de ejecución. La función `oql` toma como parámetros la referencia a una variable en la cual se almacenará el resultado, la sentencia OQL y una lista de longitud variable de expresiones C++ cuyos valores son los operandos de entrada para la consulta y se accesan por posición, precedidos del signo "$". La función regresa un valor diferente de cero si la consulta no está bien formada. Como ejemplo se muestra el código del método `ancestros` de la clase `persona`

```
Set<Ref<Persona>>Persona::ancestros(){
  Set<Ref<Persona>> resultado;

  oql(resultado, "flatten(select distinct a->ancestros
                             from a in $1c) \
          union $1c,"padres);
  return resultado;
};
```

En el ejemplo se muestra que una consulta recursiva puede escribirse fácilmente. El resultado de la cláusula `select` es un conjunto de conjuntos, con `flatten` éste se convierte en conjunto sencillo. Luego se toma su unión con el conjunto `padres`. La recursión se detiene cuando el conjunto de padres es vacío.

5 Trabajos recientes

ODMG sigue trabajando para fortalecer la versión actual del estándar y para agregarle nuevas características. Entre los desarrollos más novedosos están la creación de un grupo de trabajo en Java cuyo objetivo es el de especificar un enlace entre ODMG-93 y Java. La otra novedad es la unión entre ODMG y el subcomité ANSI X3H2 (el subcomité para SQL3) para lograr convergencia entre OQL y SQL3. Por otra parte son cada vez más los productos de conformidad con ODMG-93 que hay el mercado O_2, Objectivity/DB, ObjectStore y POET entre otros.

Referencias

[1] F Bancilhon and Guy Ferran. Object databases and the odmg standard. Technical report, O_2 Technology, 1995.

[2] R.G.G Cattel. *The Object Database Standard: ODMG-93. Release 1.2.* Morgan Kaufmann Publishers. San Francisco, Cal., 1996.

[3] R Orfali, D Harkey, and J Edwards. *The essential distributed objects, survival guide.* John Wiley, 1996.

6 Datos personales

Nombre: Amparo López Gaona

Grado: Doctoranda en la Universidad Politécnica de Madrid

Áreas de Interés: Programación orientada a objetos, bases de datos y lenguajes de programación.

e-mail: agaona@dit.upm.es