Implementación de ODL para proporcionar persistencia en SGBDRs a sistemas replicados orientados a objetos*

J.E. ARMENDÁRIZ, J.J. ASTRAIN, A. CORDOBA, J.R. G. DE MENDÍVIL y J. VILLADANGOS
Dpto. Matemática e Informática, Universidad Pública de Navarra
Campus Arrosadía, 31006 Pamplona (España)

RESUMEN

Actualmente, el desarrollo de sistemas distribuidos está creciendo de manera considerable, principalmente los sistemas de replicación geográficamente distribuidos. Uno de los mayores problemas asociados con la replicación es garantizar la persistencia de los objetos manejados por las aplicaciones remotas. Para resolver este problema, este artículo presenta un traductor que permite almacenar el estado de los objetos de manera persistente (sus atributos y relaciones univaluadas y multivaluadas) en sistemas gestores de bases de datos relacionales (SGBDRs) generando las sentencias SQL correspondientes para almacenar, seleccionar y recuperar objetos. Este traductor nos permite diseñar un sistema de almacenamiento persistente de objetos que soporte a diferentes algoritmos de consistencia, replicación y tolerancia a fallos de sistemas de replicación de objetos geográficamente distribuidos.

Palabras Claves: Persistencia, Replicación, Distribuido, SGBDR, SQL, ODL, Repositorio de Objetos, OQL.

1. INTRODUCCIÓN

La mayoría de las grandes empresas tienen sedes en diferentes ciudades, incluso en diferentes países. Obviamente, estas delegaciones necesitan acceder y almacenar persistentemente la información que es compartida entre todas ellas. Si no se es cuidadoso en el diseño de la red de la empresa, puede ocurrir que el fallo de una delegación implique la caída de todo el sistema. Para prevenir esta situación, es necesario el replicar y distribuir la información entre las diferentes sedes. En nuestro caso se ha adoptado esta solución, definiendo los datos replicados en una sede como repositorio. Esta solución debe proporcionar una vista única y un acceso uniforme en todo el sistema para aplicaciones que manejan la información almacenada en los repositorios siguiendo el paradigma de la orientación a objetos.

Hoy en día, la programación orientada a objetos constituye uno de los paradigmas más importantes en cuanto a análisis, diseño e implementación. Por otro lado CORBA permite el desarrollo de aplicaciones orientadas a objeto en entornos distribuidos.

* Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología español como proyecto CICYT número: TIC2003-09420-CO2-O2.

A menudo estas aplicaciones necesitan almacenar de manera persistente la información que manejan mediante sistemas gestores de bases de datos orientados a objetos (SGBDOO) [1,2]. Los fabricantes de SGBDOO han realizado un esfuerzo realmente importante, especialmente en estos últimos años, para que se conviertan en populares. Los más importantes constituyeron el *Object Database Management Group*, actualmente *Object Data Management Group* (ODMG) [1], que han desarrollado un modelo de datos común basado en el modelo OMG, con su lenguaje asociado de consulta llamado *Object Query Language* (OQL). El estándar ODMG-93 de bases de datos orientadas a objetos [1] tiene como objetivo garantizar la portabilidad entre sistemas, para lo cual se han definido tres interfaces: (i) *Object Definition Language* (ODL). Este lenguaje define el modelo de datos, manteniendo la compatibilidad con IDL, para la definición de objetos complejos, las relaciones entre ellos y sus métodos asociados; (ii) OQL, este lenguaje permite hacer consultas sobre los objetos anteriores, envío de mensajes a objetos y realizar reuniones y otras operaciones de tipo asociativo. Sus sintaxis es muy similar a SQL [3]; y, (iii) conexión a través de C++ y SmallTalk, define los interfaces para programar una aplicación en estos lenguajes sobre una base de datos definida en ODL.

Las SGBDOO son especialmente ventajosas en aplicaciones CAD/CAM/CAE, herramientas CASE, etc. Los fabricantes deben enfrentarse al problema de un mercado restringido a las citadas aplicaciones. Este problema ha sido superado por medio de los siguientes tres elementos, los cuales no están directamente relacionados con las tecnologías de las bases de datos: (i) el lenguaje Java, el ODMG v.2 [4] extiende los interfaces a Java; (ii) el *Unified Modeling Language* (UML), la mayor utilización de UML para el diseño y especificación de los sistemas de información una mayor comodidad para su inserción y gestión en SGBDOO que en SGBDR (un enfoque 100% orientado a objetos, desde el principio hasta el final); y, (iii) el auge del desarrollo de aplicaciones distribuidas empleando CORBA.

Las bases de datos relacionales, y sus SGBDR asociados, han evolucionado para dar soporte a objetos y reglas, además de extender SQL con nuevos tipos y operaciones. Los módulos de almacenamiento persistente son una importante mejora para la productividad y seguridad de las aplicaciones. Éstos mejoran la eficiencia en los entornos cliente-servidor. El nuevo estándar SQL-99 [5] ha tratado de converger al lenguaje ODMG y a OQL. En esta nueva evolución han influido decisivamente por un lado, JDBC y SQLJ [6] y, por otro, las bases de datos deductivas no han obtenido el resultado esperado.

La mayoría de las aplicaciones más importantes que gestionan objetos están construidas sobre SGBDRs [7]. A pesar de que éstos no han adquirido el desarrollo deseado para poder incluir los nuevos conceptos asociados, sí que ha quedado patente que son más estables y eficientes que los SGBDOOs. De hecho muchas empresas siguen utilizando SGBDRs, además de que muchas aplicaciones orientadas a objetos emplean como mecanismo de almacenamiento persistente SGBDRs.

Las técnicas de propagación de actualizaciones en entornos replicados pueden variar desde técnicas asíncronas, donde las actualizaciones se propagan cuando sea posible o periódicamente, hasta las totalmente síncronas, en donde el orden de actualización de las réplicas es total [8]. Hay diferentes aproximaciones a las técnicas de replicación de datos. Así, en [9] se propone un sistema de replicación implementado en Java mediante RMI. Este sistema proporciona comunicación entre un grupo de servidores. El sistema Aroma [10], también ofrece replicación de objetos utilizando RMI, y su diferencia principal con el anterior es el empleo de mecanismos de intercepción para ofrecer la replicación de objetos de modo transparente. También existen soluciones más genéricas mediante CORBA por medio de técnicas de agrupamiento de objetos [11]. Estos trabajos adolecen de mecanismos de almacenamiento persistente para las instancias creadas.

El objetivo de este trabajo es proporcionar un soporte de almacenamiento persistente en un entorno de replicación llamado COPLA (*Common Object Programmer Library Access*) [12]. Este sistema proporciona a los desarrolladores de aplicaciones una única capa que engloba y simplifica el acceso a un repositorio de objetos compartido, cuyas características principales son: la unicidad, la consistencia y la alta disponibilidad. La novedad de esta arquitectura es que se proporciona un servicio de almacenamiento persistente de objetos sobre una arquitectura replicada, llamada *Uniform Data Store* (UDS) [13], garantizando un comportamiento transaccional y la posibilidad de acceder al estado de los objetos bien a través de consultas específicas o por la invocación directa de métodos sobre los objetos. Otras características son la posibilidad de recuperar los sitios cuando se producen caídas de sitios o la posibilidad de modificación de una aplicación previamente definida en COPLA, sin que implique la pérdida de los datos previamente almacenados.

El principal objetivo del sistema de almacenamiento persistente es proporcionar una única capa de software que aísle de los detalles de almacenamiento persistente al resto de componentes. Como medio de almacenamiento se emplea un SGBDR, debido a su madurez tecnológica y a la familiaridad que tienen con ella tanto los administradores de los sistemas como los usuarios finales, además de permitir la migración de la información almacenada en bases de datos a las aplicaciones cliente orientadas a objeto.

Es importante destacar que la arquitectura proporciona un mecanismo de replicación siguiendo el paradigma de la orientación a objetos mientras que el sistema de persistencia emplea el modelo entidad-relación [14,15,16]. Este desajuste debe ser solventado mediante la especificación de un lenguaje de definición de objetos, llamado *Globdata Object Definition Language* (GODL), que establece como deben ser definidas las clases, los atributos y las relaciones (univaluadas y multivaluadas), así como su mapeo relacional sobre el SGBDR. Se debe definir además como se representa el estado del objeto almacenado persistentemente en la aplicación.

Este trabajo presenta un traductor que hace posible el almacenamiento y la recuperación del estado de los objetos en SGBDRs. Comenzando por la definición de un módulo GODL que contendrá la definición de los objetos, el traductor genera las sentencias SQL que permite generar la estructura de tablas adecuada para poder almacenar, actualizar o borrar el estado de los objetos pertenecientes al módulo actual.

El traductor debe resolver dos problemas esenciales para asegurar el almacenamiento persistente en SGBDRs: (i) La gestión del mapeo, es decir, cómo se almacenan los objetos en la base de datos, gestionar los atributos, las relaciones, la herencia, etc.; (ii) la gestión de los metadatos, como se debe emplear una capa de software para aislar los detalles de la implementación en el SGBDR, hay que definir la meta-información que facilite al UDS para localizar correctamente los objetos, así como la gestión de la consistencia ya que estamos dentro de un entorno replicado.

El resto del artículo esta estructurado de la siguiente manera, la Sección 2 específica los trabajos relacionados acerca del almacenamiento persistente de objetos. La arquitectura del sistema se explica brevemente en la Sección 3. La Sección 4 introduce la gramática GODL. La arquitectura del traductor se muestra en la Sección 5. El mapeo de cada uno de los elementos de GODL sobre el SGBDR se describe en la Sección 6. El conjunto de metadatos necesarios para su correcto funcionamiento están contenidos en la Sección 7. Este trabajo finaliza con una sección dedicada a las conclusiones.

2. TRABAJOS RELACIONADOS

En [7] se muestra el diseño y la implementación de un sistema de almacenamiento persistente para guardar el estado de los objetos en medios no volátiles como discos duros o bases de datos, relacionales u orientadas a objetos. Existen implementaciones para entornos concurrentes [17,18]. Otros emplean CORBA para garantizar la persistencia de los objetos [19]. En [20] se trata la persistencia en agentes móviles, mientras que en [21] se describe la persistencia de objetos Java para aplicaciones de *intranet* en empresas o aplicaciones *web* específicas. Nuestra implementación sigue varios aspectos de los trabajos anteriores, aunque en todo lo relativo al almacenamiento de colecciones y relaciones multivaluadas no hemos conseguido una referencia con la que comparar lo aportado en este artículo.

3. ARQUITECTURA COPLA

La arquitectura COPLA está dividida en tres capas, tal y como se muestra en la Figura 1. Estas capas están implementadas en Java y pueden residir en diferentes máquinas puesto que emplean un ORB para interactuar entre ellas.

Si nos basamos en la estructura de la Figura 1 y siguiendo una aproximación del nivel superior a inferior nos encontramos la primera capam, denominada *library*, dicha capa contiene una serie de paquetes Java que proporciona toda la lógica de programación necesaria para el desarrollo de las aplicaciones COPLA en GODL [1,22]. Precisamente la definición de estas aplicaciones y su almacenamiento persistente es lo que se va a describir a lo largo de este artículo. De este modo las clases (*class*) están definidas dentro de un esquema o módulo GODL (*GODL schema*), el cual define un repositorio de objetos. Los objetos creados dentro de un repositorio de objetos pueden ser accedidos concurrentemente dentro del contexto de una transacción distribuida [12]. Los usuarios a su vez emplean un subconjunto de OQL [22] para obtener las referencias a estos objetos distribuidos. Una vez que se obtienen estas referencias, la aplicación puede modificar éstos u obtener nuevas referencias a otros objetos mediante los atributos (*attribute*) o relaciones (*relationship*) de los objetos obtenidos inicialmente. Una vez que la aplicación ha finalizado solicitará la consumación de la transacción (*commit*).

La capa denominada *COPLA manager* es el componente clave de la arquitectura, entre sus responsabilidades está la de gestionar una caché de objetos y la consistencia de los mismos entre las diferentes réplicas, las cuales están situadas en diferentes sitios o nodos. El *COPLA manager* necesita un protocolo específico (o gestor de la consistencia, C*onsistency Manager*, como se muestra en la Figura 1) que defina una serie de reglas acerca de la actualización de las réplicas en un determinado orden, de hecho en COPLA ya tiene implantado varios protocolos de consistencia [12,23,24]. Así pues éstos deben determinar la existencia de conflictos entre nodos distintos que tratan de acceder concurrentemente a un mismo objeto. La elección del mejor protocolo depende de la topología de red y de la carga de trabajo de la aplicación empleada. Estos protocolos implementan un interfaz común, con lo cual el intercambio de un protocolo a otro se realiza de manera sencilla y COPLA se puede configurar de acuerdo con las características con las cuales se está ejecutando. Finalmente, esta capa es la que se encarga, de manera exclusiva, de la comunicación entre los nodos que componen la arquitectura COPLA.

La última capa mostrada en la Figura 1 es la que proporciona la persistencia de los objetos creados en COPLA mediante su almacenamiento en una SGBDR, su nombre es *Uniform Data Store* (UDS) [13]. Esta capa implementa un interfaz UDS-API a través del cual se pueden acceder y almacenar los objetos. De este modo se aíslan los detalles de almacenamiento en un sistema relacional al resto de los componentes de la

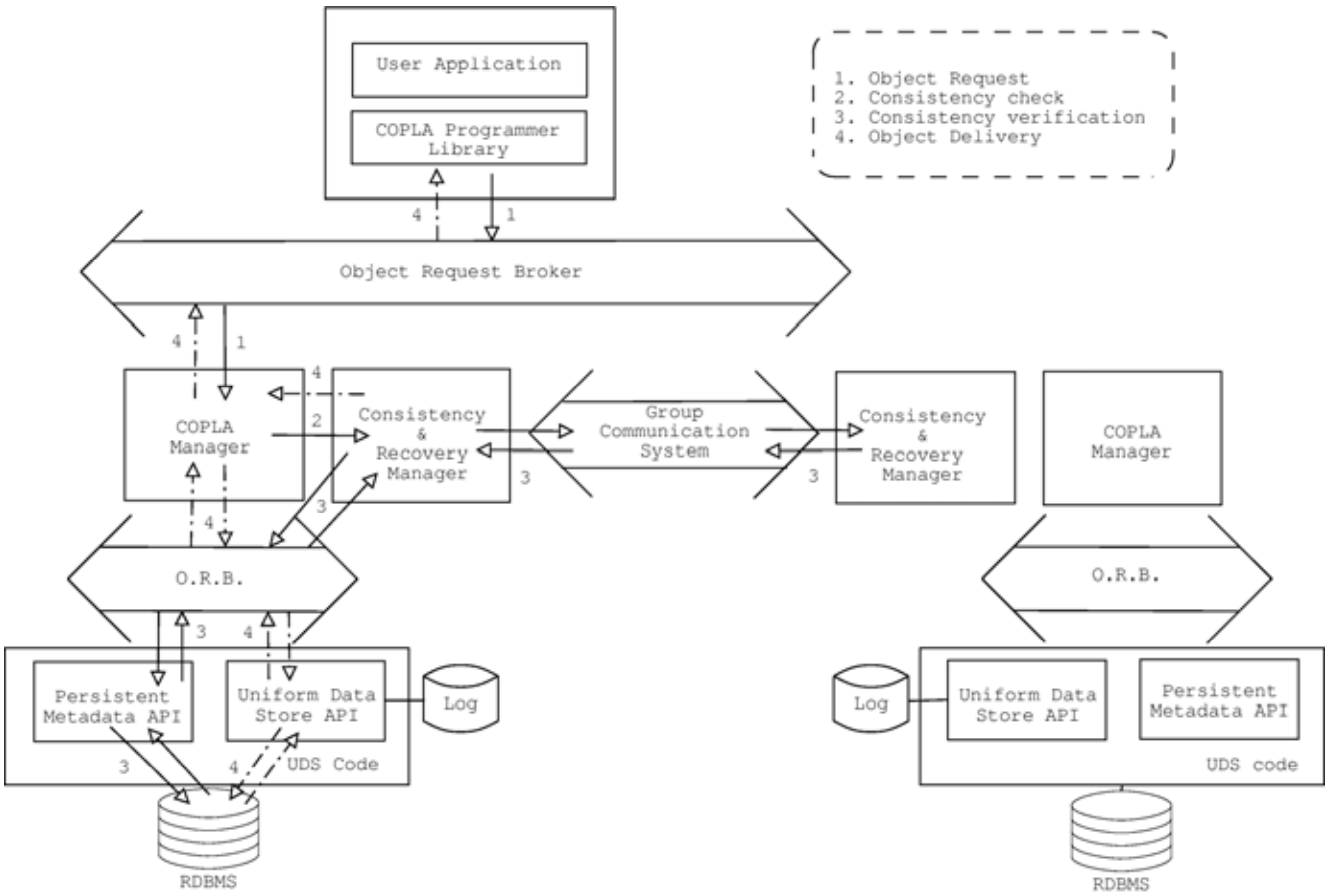

Figura 1: La arquitectura COPLA.

arquitectura COPLA. Como se ha citado anteriormente, los objetos en COPLA son accedidos en el contexto de una transacción distribuida que se corresponde con una transacción dentro del SGBDR con el nivel de aislamiento adecuado. El UDS se encarga de traducir todas las *queries* realizadas por el usuario a sentencias SQL estándar [22]. Finalmente, el UDS se emplea para almacenar de manera persistente la información de control que necesita cada uno de los protocolos para su correcto funcionamiento.

4. GRAMÁTICA GODL

GODL es un subconjunto del estándar *Object Definition Language* [1]. La principal restricción es que no se permiten colecciones anidadas, ya que no es uno de los principales objetivos de nuestro trabajo. En esta sección se describirá la gramática GODL, cuyos principales elementos son los módulos, las clase, los atributos y las relaciones.

```
1  module mod_name {
2   class class_name1 {
3     attribute long id_number;
4     attribute string<10> name;
5     attribute class_name3 att_class3;
6     attribute set<class_name2> att_class2;
7     ....
8     relationship type1 class_name2 identifier1
9     inverse class_name2::identifier2;
10    ....
11  };
12  class class_name2 {
13    ....
14    relationship type2 class_name1 identifier2
15    inverse class_name1::identifier1;
16    ....
17  };
18  class class_name3 extends class_name1{
19    ....
20  };
21 }
```

Figura 2: Módulo GODL genérico.

Las clases están definidas dentro de un módulo, en ella se especifican un conjunto de atributos y relaciones, así como la posible herencia de otra clase previemente definida. Los atributos, los cuales pueden ser multivaluados (*arrays*, conjuntos, bolsas y listas), son como propiedades que pueden ser de tipos básicos, *strings* o instancias de otras clases. Las relaciones son asociaciones entre instancia de la misma o diferentes clases, univaluadas o multivaluadas. En la Figura 2 se muestra un ejemplo de un esquema GODL.

5. TRADUCTOR GODL

El traductor GODL es un compilador cuya entrada es un esquema GODL y su salida es un conjunto de sentencias SQL que permite almacenar el estado de los objetos en un SGBDR. Los bloques que constituyen el traductor son: un analizador léxico y un analizador sintáctico. El analizador sintáctico, toma como entrada los *tokens* enviados por el analizador léxico, para ir construyendo la correspondiente tabla de símbolos. Después genera el mapeo correspondiente al módulo compilado. El traductor es un compilador de dos etapas. En la primera, el fichero que contiene el módulo GODL es analizado sintácticamente, y simultáneamente se va generando la tabla de símbolos. En la segunda etapa, se generan las sentencias SQL correspondientes. JLex [25] y CUP [26] son el analizador léxico y el generador de *parser*s empleados para la realización del compilador respectivamente.

Inicialmente, el traductor genera un fichero de extensión *lex* que contiene el listado de las palabras reservadas de GODL, además de otro (con la extensión *cup*) que contiene la gramática GODL. JLex va obteniendo *token* del módulo que va enviando al analizador sintáctico, el CUP. Éste debe encontrar la correspondiente regla de producción gramatical para estos *tokens* y ejecutar el código correspondiente a estos segmentos. Si todo funciona correctamente se va construyendo la tabla de símbolos, en caso contrario se lanzará una excepción. Un módulo independiente, denominado *mapping*, extrae la información de la tabla de símbolos y genera el correspondiente mapeo, las sentencias SQL y los *triggers* y funciones para mantener la integridad referencial y la gestión de las colecciones.

6. TRADUCTOR GODL

El propósito de nuestra propuesta de mapeo es almacenar el estado de los objetos definidos en un módulo, como el descrito en la Figura 3, en un SGBDR. En esta sección se explicará la traducción del módulo a su equivalente en el modelo entidad-relación. En COPLA se emplea un identificador de objeto (*Globdata Object Identifier*, GOID) para poder localizar a un objeto dentro del sistema. Este valor se almacena en una tabla de metainformación llamada `GOIDS`, sobre la cual todas las tablas hacen referencia como clave extranjera, tal y como se explica en lo que resta de sección.

Mapeo de atributos univaluados

El mapeo de atributos univaluados (líneas 3-5 de la Figura 2) se realiza de distinta manera dependiendo de si son literales u objetos como se muestra en la Figura 3. Para cada clase definida en el módulo se genera una tabla que contiene los siguientes campos: `GOID`, que identifica al objeto y hace referencia a la tabla `GOIDS`; y, tantas columnas como atributos literales univaluados, enumerados o de tipo *string* (estos son de longitud fija, especificada por el usuario, y definidos como `VARCHAR(length)`).

Los tipos válidos de literales son: `char`, `boolean`, `octet`, `short`, `unsigned short`, `long`, `long long`, `float`, `double`, `date`, `time` y `timestamp`. Dependiendo del tipo de SGBDR puede suceder que no exista un reflejo de cada uno de esos tipos en tipos de la base de datos; en general, todos estos tipos tienen su equivalente en el SGBDR. Si existen atributos de tipo `unsigned short` o `enum` se generan las restricciones correspondientes sobre los campos. GODL permite la definición de atributos literales univaluados clave que se traducen en el SGBDR en la especificación de claves primarias.

Los atributos objeto univaluados se mapean en tablas separadas, una por cada atributo definido en la clase respectiva. El nombre de esta nueva tabla se construye del siguiente modo: el nombre de la clase donde el atributo ha sido definido,

seguido de la cadena "_O_" y finalizado por el nombre del atributo. Esta nueva tabla incluye únicamente dos campos: el primero contiene el GOID del objeto de la clase donde se ha definido el atributo y, el segundo, el GOID del objeto perteneciente a la clase agregada. Ambos campos hacen referencia al campo GOID de la tabla de la clase a la que pertenece cada una de ellas, tal y como se muestra en la Figura 3. Así, la eliminación de uno de ellos provoca el borrado en cascada de todos los registros de las tablas donde estuvieren almacenados.

```
CREATE TABLE CLASS_NAME(OBJ_ID VARCHAR REFERENCES
AGGREG(OBJ_ID) ON UPDATE CASCADE ON DELETE CASCADE,
LITERAL_TYPE_1 <RDBMS_TYPE>, ... LITERAL_TYPE_n
<RDBMS_TYPE>);

CREATE TABLE CLASS_NAME_O_IDENT(CLASS_NAME VARCHAR
REFERENCES CLASS_NAME(GOID) ON UPDATE CASCADE ON
DELETE CASCADE, IDENT VARCHAR REFERENCES
CLASS_NAME_2(GOID) ON UPDATE CASCADE ON DELETE
CASCADE, PRIMARY KEY(CLASS_NAME), UNIQUE(IDENT));
```

Figura 3: Mapeo de atributos univaluados.

Mapeo de atributos multivaluados

GODL permite la definición atributos multivaluados, bien sean objetos o literales, en los siguientes tipos de colecciones: bolsas (BAG), conjuntos (SET) y listas (LIST) (ver la línea 6 de la Figura 2). Todas las colecciones se almacenan en tablas diferentes, como se puede observar en la Figura 4. El nombre de la tabla es el nombre de la clase seguido de "_B_", "_S_" o "_L_" dependiendo del tipo de colección, aunque en la Figura 4 hemos puesto "_COLLECTION_", y el nombre del atributo.

```
CREATE TABLE CLASSNAME_COLLECTION_IDENT(CLASSNAME
VARCHAR REFERENCES CLASSNAME(GOID) ON UPDATE CASCADE
ON DELETE CASCADE, [IDENT RDBMS_TYPE  NOT NULL,| IDENT
VARCHAR NOT NULL REFERENCES CLASS_NAME2 ON UPDATE
CASCADE ON DELETE CASCADE], [UNIQUE(IDENT), PRIMARY
KEY(CLASSNAME, IDENT)] [CARDINAL_NUMBER INT4,
CHECK(CARDINAL_NUMBER>=0)]);

CREATE TABLE CLASSNAME_A_IDENT(CLASSNAME VARCHAR
REFERENCES CLASSNAME(GOID) ON UPDATE CASCADE ON DELETE
CASCADE, [IDENT RDBMS_TYPE  NOT NULL,| IDENT VARCHAR
NOT NULL REFERENCES CLASS_NAME2 ON UPDATE CASCADE ON
DELETE CASCADE], DIMENSION_1 INT4,
CHECK(DIMENSION_1>=0 AND DIMENSION_1< n), PRIMARY
KEY(CLASSNAME, DIMENSION_1));
```

Figura 4: Mapeo de atributos multivaluados.

En el caso en que se defina un atributo del tipo bolsa, se crea una nueva tabla que contendrá dos campos: el primero contiene los identificadores de objeto que son dueños de la bolsa, es decir, contiene una referencia al campo GOID de la clase donde se ha definido ese atributo. El segundo campo, cuyo nombre coincide con el nombre del atributo, de esta nueva tabla varía dependiendo de si se ha definido una bolsa que contiene literales u objetos; para los literales se define el tipo literal correspondiente, y en el caso de objetos, se procede de manera análoga a la definición del primer campo de esta tabla. Finalmente, no se impone ninguna restricción acerca de los elementos que se pueden almacenar en la bolsa, ya que contiene elementos repetidos sin ningún tipo de orden.

Si se especifica un conjunto que va a contener como elementos atributos literales, se construye la nueva tabla cuyos dos campos son exactamente iguales al de la bolsa. Se añaden dos restricciones para garantizar que no se introduzcan elementos repetidos.

Finalmente, la especificación de listas se realiza exactamente igual que las dos anteriores, pero se incluye un nuevo campo para que establezca el orden de los elementos almacenados. Dado que GODL permite la inserción y borrado de elementos en cualquier posición de la lista, se han creado una serie de *triggers* para que cuando se añada (o borre) un elemento de la lista se regenere de manera automática las posiciones asociadas a cada elemento en la lista.

Otro atributo multivaluado considerado en GODL es el ARRAY, que se mapea en una tabla distinta y contiene al menos tres campos: La primera contiene el objeto propietario del *array*, la segunda el valor asociado a esa posición del *array* (ya sea un literal o un objeto) y el resto serán los campos que indican la posición de ese valor dentro de la dimensión del *array*. Cuando se crea un array asociado a un objeto, se ha definido una función que genera todas las posiciones posibles para el *array* con todos los elementos nulos, así como otra función, activada por un *trigger*, para generar de nuevo el registro asociado a una posición del *array* cuando se borra el elemento contenido en ella.

```
CREATE TABLE CLASS_NAME1_IDENT1(CLASS_NAME1_IDENT1
VARCHAR REFERENCES CLASS_NAME1(goid) ON UPDATE CASCADE
ON DELETE CASCADE, CLASS_NAME2_IDENT2 VARCHAR
REFERENCES CLASS_NAME2(goid) ON UPDATE CASCADE ON
DELETE CASCADE);
Additional clauses added depending on:
1-1.- To assure that one object is only related with
another:
  PRIMARY KEY(CLASS_NAME1_IDENT1),
UNIQUE(CLASS_NAME2_IDENT2)
1-SET.- Un objeto puede estar relacionado con varios
objetos no repetidos.
  PRIMARY KEY(CLASS_NAME1_IDENT1, CLASS_NAME2_IDENT2),
UNIQUE(CLASS_NAME1_IDENT1)
1-LIST.- Se añade un Nuevo campo para mantener el
orden de las relaciones entre los objetos.
  IDENT2_CARDINAL INT4, CHECK(IDENT2_CARDINAL >=0),
UNIQUE(CLASS_NAME1_IDENT1)
SET-SET.- Un conjunto de objetos pueden estar
relacionados con otro conjunto de objetos.
  PRIMARY KEY(CLASS_NAME1_IDENT1, CLASS_NAME2_IDENT2)
SET-LIST.- Un conjunto de objetos están relacionados
con una lista de objetos, para ello es necesario
añadir el siguiente campo.
  IDENT2_CARDINAL INT4, CHECK(IDENT2_CARDINAL >=0),
UNIQUE(CLASS_NAME1_IDENT1, CLASS_NAME2_IDENT2)
LIST-LIST.- Se añaden dos columnas adicionales.
  IDENT1_CARDINAL INT4, CHECK(IDENT1_CARDINAL >=0),
IDENT2_CARDINAL INT4, CHECK(IDENT2_CARDINAL >=0)
```

Figura 5: Mapeo de asociaciones.

Mapeo de asociaciones

Este tipo de mapeo es más sencillo que el de los atributos, ya que las relaciones pueden verse como asociaciones dentro del modelo entidad-relación. Asumimos que cada relación GODL es opcional pudiendo ser ésta univaluada o multivaluada (la declaración de una relación, *relationship*, se muestra en la Figura 2 en la líneas 8-9 y 14-15). Los tipos de relaciones válidas se incluyen en la Figura 5. Si en una relación se define en alguno (o en ambos) de los sentidos una lista, el compilador genera los *triggers* y las funciones adecuados para su gestión. El nombre de la tabla está compuesto por la clase en que primero se haya definido la relación en el esquema, los nombres de los campos son los de las relaciones en ambos

sentidos, y a lo sumo aparecerán dos campos adicionales si alguno (o ambos) de los sentidos contiene una lista.

Los tipos de relaciones considerados no son todas las posibles combinaciones de univaluados y los distintos tipos de colección entre si, ya que se generan asociaciones redundantes. Esta circunstancia se verá más clara con un ejemplo, si se ha definido una relación de uno con una bolsa de objetos, significa que si hay un elemento repetido dentro de la bolsa implica que está asociado *n* veces, siendo *n* el número de repeticiones de ese elemento, cuando, en realidad, no hay diferencia entre estar asociado una vez o *n*. De esta reflexión y similares establecimos el conjunto de asociaciones válidas entre objetos pertenecientes a la misma clase o distintas clases que se muestra en la Figura 5.

7. METADATOS

Los metadatos generados por el compilador se pueden dividir en dos conjuntos dependiendo de su funcionalidad. El primer conjunto sirve para localizar los diferentes elementos que componen un esquema GODL. El segundo contiene la información necesaria para el correcto funcionamiento de los protocolos que gestionan la consistencia del estado de los objetos almacenados en la base de datos. Además dado que existen varios protocolos implantados en COPLA se deben generar las tablas de metainformación asociados a cada uno de ellos.

El primer conjunto de metadatos que hemos mencionado contiene información acerca de la herencia, los atributos y las relaciones definidas para un módulo. Toda esta información es necesaria para que la capa que dé soporte a la persistencia de los objetos pueda llevar a cabo correctamente todas las operaciones sobre el estado de un objeto. Los metadatos se desglosan en cuatro tablas: `GOIDS`; `HIERARCHY`; `ATTRIBUTES`; y, `RELATIONSHIPS` (véase la Figura 6).

Comenzaremos por la tabla `GOIDS`. Esta tabla contiene todos los objetos definidos en el repositorio y lleva el control de la agregación de los objetos. Contiene tres campos: el primero para el identificador del objeto, el segundo es para el identificador del objeto al que está agregado y el último contiene la multiplicidad de la citada agregación. De esta manera, cuando se desea borrar un objeto del repositorio, se borra el registro asociado a su identificador en esta tabla, esto genera un borrado en cascada de todos los registros que contengan ese identificador de objeto en la base de datos, sin olvidar que los *triggers* reconstruirán las listas y *arrays* alterados por este borrado.

Hasta ahora no se hablado nada de la jerarquía asociada a las clases, dicha información son metadatos asociados a una nueva tabla en la base de datos, llamada `HIERARCHY`. Esta tabla contiene por lo menos tantos registros como clases se hayan definido en el esquema. En el caso de que alguna clase herede de alguna otra, esta tabla contendrá tantos registros de la primera como profundidad tenga su árbol jerárquico.

La tercera tabla de información (`ATTRIBUTES`) contiene todos los atributos definidos en el repositorio. Dicha tabla contiene toda la descripción de ese atributo, si es literal, multivaluado, la clase donde ha sido definido, su nombre, etc. De tal manera que el UDS puede localizar los atributos de un objeto sin ningún tipo de problema.

La última tabla (`RELATIONSHIPS`) contiene la información asociada a las relaciones definidas en el repositorio. Como en el caso de los atributos esta tabla sirve para localizar y determinar las relaciones asociadas en la base de datos: clases relacionadas, los nombres de las relaciones en ambos sentidos, el tipo de la relación, etc.

El segundo conjunto de metainformación corresponde al almacenamiento de metadatos asociados a la replicación del estado de los objetos en diferentes asociados viene reflejado únicamente a modo de muestra en la Figura 6; su forma depende del protocolo de consistencia considerado, en este caso consideramos el presentado en [12]. Estas tablas contienen, en general, información acerca de los objetos almacenados en el repositorio, su versión, en que nodo fueron creados, etc. Esta información la emplea el protocolo para saber si está accediendo a información obsoleta y gestionar recuperaciones ante caídas de nodos.

```
CREATE TABLE GOIDS(GOID VARCHAR NOT NULL PRIMARY KEY,
IS_AGGREGATED_TO VARCHAR REFERENCES AGGREG(GOID) ON
UPDATE CASCADE ON DELETE CASCADE, COUNTER_REFERENCES
INT4 DEFAULT 0, CHECK(COUNTER_REFERENCES>=0));

CREATE TABLE HIERARCHY(CLASS VARCHAR(14), SUPERCLASS
VARCHAR(14));

CREATE TABLE ATTRIBUTES(ATTRIBUTE_KEY VARCHAR,
CLASS_NAME VARCHAR, COLLECTION_TYPE VARCHAR(5), TYPE
VARCHAR, ARRAY_SIZE VARCHAR, PRIMARY
KEY(ATTRIBUTE_KEY, CLASS_NAME));

CREATE TABLE RELATIONSHIPS(RELATIONSHIP_KEY VARCHAR,
CLASS_NAME VARCHAR,
INVERSE_NAME VARCHAR, INVERSE_CLASS_NAME VARCHAR,
COLLECTION_TYPE VARCHAR(4), TABLE_NAME VARCHAR,
PRIMARY KEY(RELATIONSHIP_KEY, CLASS_NAME));

CREATE TABLE ITIPERMETADATA (GOID VARCHAR REFERENCES
GOIDS(GOID) ON UPDATE CASCADE ON DELETE CASCADE
DEFERRABLE, VERSION INT4, NODE_ID INT4, SID VARCHAR,
READ BOOLEAN);
```

Figura 6: Mapeo de metadatos.

8. CONCLUSIONES

En este artículo se ha presentado un traductor que permite a una aplicación orientada a objetos almacenar persistentemente el estado de los objetos definidos en ella sobre un SGBDR. El traductor se ha implementado con Java, JLex y CUP. Para ello se ha definido el lenguaje GODL, un subconjunto del estándar propuesto por ODMG, que establece cómo deben ser definidas las aplicaciones en el entorno replicado, a las que hemos llamado módulos. Una de las principales diferencias es que no permitimos la definición de colecciones anidadas.

Un módulo se mapea en una nueva base de datos, las clases con todos sus atributos literales univaluados se mapean en una tabla. El resto de atributos, así como las relaciones se mapean en tablas diferentes. La gestión de colecciones se realiza mediante mecanismos internos del SGBDR, como son claves primarias *triggers* y funciones. En la literatura no hemos podido (o sabido) encontrar mapeos para poder establecer comparaciones con los tipos multivaluados y las relaciones. El

traductor genera meta-información adicional para que una capa de software instalada por encima del SGBDR sea capaz de poder realizar operaciones sobre el estado de los objetos. Además en el proceso de traducción se ha seguido el estándar SQL para hacerlo lo más independiente posible del SGBDR que se vaya a emplear.

9. REFERENCIAS

[1] R.G.G. Cattell y D.K. Barry. **The Object Data Standard: ODMG 3.0**. Morgan Kauffmann Publishers. USA, 2000.

[2] A. de Miguel y M. Piattini. **Fundamentos y modelos de Bases de Datos**. Ediciones Ra-Ma, Segunda edición. Madrid, España, 1999.

[3] C.J. Date y H. Darwen. **A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL**. Addison-Wesley Publishers. USA, 1997.

[4] R.G. Cattell y D.K. Barry. **The Object Database Standard ODMG 2.0**. Morgan Kauffmann Publishers. USA, 1997.

[5] P. Gulutzan y T. Pelzer. **SQL-99 Complete, Really**. CMP Books. USA, Marzo 1999.

[6] Welcome to www.SQLj.org! Your Online Resource For Information About SQLj: `http://www.sqlj.org`

[7] J. Chen, Q.-M. Huang y A.S.M. Sajeev. "Interoperability Between Object-Oriented Programming and Relational Systems". **In Proceedings of the 6th International Hong Kong Database Workshop**, Hong Kong, Marzo 1995.

[8] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme y G. Alonso. "Understanding Replication in Databases and Distributed Systems". **In Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)**, Taipei, Taiwan, R.O.C., págs. 264-274, April 2000.

[9] A. Baratloo, P.E. Chung, Y. Huang, S. Rangarajan y S. Yajnik. "Filterfresh: Hot Replication of Java RMI Server Objects", **In Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)**, New Mexico, USA, Abril, 1998.

[10] N. Narasimhan, L.E. Moser y P.M. Melliar-Smith. "Transparent Consistent Replication of Java RMI Objects", **In Proceedings of International Symposium on Distributed Objects and Applications**, Antwerp, Belgium, Septiembre 2000.

[11] P. Felber, R. Guerraoui y A. Schiper. "Replication of CORBA objects". **In S. Krakowiak and S.K. Shrivastava, editors, Recent Advances in Distributed Systems, volume 1752 of LNCS**. Springer Verlag, 2000.

[12] F.D. Muñoz-Escoi, J.M. Bernabéu-Aubán y L. Irún-Briz. "Globdata: Consistency Protocols for Replicated Architectures", **Youth Forum In Computer Science and Engineering**, Valencia, España, Noviembre 2001.

[13] J.E. Armendáriz, J.J. Astrain, A. Córdoba, J. Villadangos y J.R. González de Mendívil. "A persistent object storage service on replicated architectures". **In Proc. of VI Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS 03)**, Asunción, Paraguay, págs. 133-144, Abril 2003.

[14] W.G. Keller. "Mapping objects to tables: A pattern language", **In Proc. of the 1997 European Pattern Languages of Programming Conference** (Siemens Technical Report 120/SW1/FB), Irsee, Germany, 1997.

[15] S. Ambler. **Agile Database Techniques - Effective Strategies for the Agile Software Developer**. John Wiley & Sons. USA 2003.

[16] K. Brown y B.G. Whitenack. "Crossing Chasms, A Pattern Language for Object-RDBMS Integration", **Editors: J.M. Vlissides, J.O. Coplien and N.L. Kerth in Pattern Languages of Program Design 2**, Addison-Wesley, USA 1996.

[17] M.P. Herlihy y J.M. Wing. "Linearizablility: A Correctness Condition for Concurrent Objects", **ACM Trans. Programming Languages**, vol. 12, no. 3, págs. 464-492, Julio 1990.

[18] P. Ferreira, M. Shapiro y otros. "PerDiS: design, implementation, and use of a PERsistent DIstributed Store", **Recent Advances in Distributed Systems, Lecture Notes in Computer Science**, Springer-Verlag, vol. 1752, no. 18, págs. 427-452, Febrero 2002.

[19] C. Mayers. "ANSAwise - Persistent data storage with CORBA", Citrix Systems, Cambridge, UK, Abril 1996.

[20] M. Mira da Silva y M. Atkinson. "Combining mobile agents with persistent systems: opportunities and challenges". **2nd (ECOOP) Workshop on Mobile Object Systems**, Linz, Austria, págs. 36-40, 1996.

[21] Bringing a new dimension to Java through easy access enterprise data: `http://www.javera.com`

[22] J.E. Armendáriz, J.J. Astrain, A. Córdoba y J. Villadangos. "Implementation of an object query language for replicated architectures". **Actas de las VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 03)**, Alicante, Spain, págs. 441-450, Noviembre 2003.

[23] L. Rodrigues, H. Miranda, R. Almeida, J. Martins y P. Vicente. "Strong replication in the globdata middleware". **In Proc. of Workshop on Dependable Middleware-Based Systems (Supplemental Volume of the 2002 Dependable Systems and Networks Conference)**, Washington D.C., USA, pages G96-G104, Junio 2002.

[24] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, y P. Vicente. "The GlobData fault-tolerant replicated distributed object database". **In Proc. of the First Eurasian Conference on Advances in Information and Communication Technology**, Teheran, Iran, Octubre 2002.

[25] JLex: A Lexical Analyzer Generator for Java (TM): http://www.cs.princeton.edu/~appel/modern/java/JLex/

[26] CUP Parser Generator for Java: http://www.cs.princeton.edu/~appel/modern/java/CUP/