Inclusión de Vistas en ODMG

Ginés García Mateos, Jesús Joaquín García Molina, Maria José Ortín Ibáñez

Departamento de Informática y Sistemas, Universidad de Murcia,
30.071 Campus de Espinardo, Murcia, España
{ginesgm, jmolina, mjortin}@um.es

Resumen. Las vistas son una importante funcionalidad proporcionada por los sistemas de bases de datos relacionales que no ha sido posible trasladar satisfactoriamente a los sistemas de bases de datos orientadas a objetos. En este trabajo presentamos una propuesta para incluir vistas en el estándar ODMG, extendiendo el modelo de objetos, el lenguaje ODL y la ligadura (*binding)* a C++. En el modelo de vistas propuesto, i) las vistas son un nuevo modo de definición de tipos que se añade a clases, interfaces y literales, ii) se introduce la relación *ISVIEW*, para especificar la derivación de una vista de un tipo base, iii) la identidad de una instancia de la vista es la misma que la de la correspondiente instancia de la clase base, y iv) es posible el *binding* a cualquier lenguaje orientado a objetos.

1 Introducción

Uno de los factores que ha impedido que los sistemas de bases de datos orientadas a objetos hayan alcanzado el éxito esperado ha sido la dificultad para proporcionar toda la funcionalidad que ofrecen los sistemas relacionales [2]. Las vistas han sido una de las funciones básicas de los sistemas relacionales que no ha sido posible trasladar a los sistemas orientados a objetos (OO). En la pasada década, se idearon diferentes propuestas de mecanismos de vistas OO [5,6,7,8,9,10] sin alcanzarse un consenso y además casi todas ellas se basaron en modelos de datos OO alejados de los modelos soportados por los sistemas comerciales y del estándar ODMG [3]. De hecho, ODMG no incluye todavía el concepto de vista, a pesar de ser bien reconocido como algo indispensable para un sistema de bases de datos OO.

En este trabajo presentamos una propuesta para la inclusión de vistas en ODMG que ofrecen la misma funcionalidad de las vistas relacionales. Esto requiere introducir extensiones en casi todos los componentes que constituyen el estándar: el modelo de objetos, ODL, y la ligadura (*binding*) a C++, Smalltalk y Java. El modelo de objetos del ODMG es extendido de modo que las vistas son tipos que se definen a través de una relación *ISVIEW* aplicada sobre el tipo (interfaz, clase o vista) base. Las instancias de una vista mantienen la identidad de las instancias del tipo base, de modo que una vista no genera nuevas instancias.

La estructura de este artículo es la siguiente. En el apartado 2 se introducen las propiedades del modelo de vistas OO establecido. A continuación, se presenta cómo es posible extender el modelo de objetos y el ODL del ODMG para permitir la definición de vistas. En el apartado 4 se describe cómo son traducidas a clases C++ las

especificaciones ODL extendidas con vistas. Por último, en el apartado 5, se presentan las conclusiones y trabajo futuro.

2 Propiedades del Modelo de Vistas Orientadas a Objetos

En [4,5] se señalan los dos principales problemas a resolver en el diseño de un modelo de vistas OO: posición de las vistas en el esquema de la base de datos (jerarquía de clases) y naturaleza de las instancias de una vista. Las decisiones de diseño en cada una de estas dos dimensiones están influenciadas por la funcionalidad de las vistas y el modelo de datos elegidos.

Como señalamos en el apartado anterior, en este trabajo proponemos un modelo de vistas para el estándar ODMG [3], por lo que el modelo de datos considerado será el modelo de objetos de ODMG. Por otra parte, establecemos que las vistas OO puedan utilizarse para las mismas funciones que las vistas relacionales: i) definición de esquemas externos, ii) autorizaciones basadas en el contenido (protección de datos) y iii) simplificación de consultas. Sin embargo, a diferencia de otros modelos [5,7,10] no introducimos la posibilidad de utilizar las vistas para simular la evolución del esquema, ya que esto complica excesivamente el modelo de vistas, al ser necesarias vistas que generen nuevas instancias y que puedan añadir atributos no derivados de datos existentes (*capacity augmenting views*). Creemos que esta funcionalidad es innecesaria para las vistas ya que la evolución del esquema es manejada mejor con mecanismos de versiones.

La definición de esquemas externos sobre los que es posible desarrollar aplicaciones, exige que las vistas tengan la misma naturaleza que las clases o interfaces, del mismo modo que las vistas relacionales pueden tratarse como tablas. Así, una vista podrá usarse en cualquier contexto donde puede aparecer una clase, por ejemplo, suponiendo tipado estático, el tipo de un atributo de una clase o el tipo de una variable en una aplicación que accede a la base de datos.

Como una vista puede actuar como un tipo, entonces es necesario determinar la posición de la vista dentro de la jerarquía de tipos, teniendo en cuenta la semántica de la relación *ISA*. La tendencia en los modelos de vistas ha sido considerar una jerarquía de vistas separada de la jerarquía de clases, introduciendo una relación de derivación entre la vista y su clase base [5,6,8,10]. En nuestro caso, las vistas participan en el esquema de la base de datos al mismo nivel que los demás tipos, lo que significa que no existen esquemas separados para vistas y para clases, aunque sí introducimos la mencionada relación de derivación y la de herencia entre vistas.

Un principio que creemos importante mantener es la independencia del esquema conceptual en relación al mecanismo de vistas: los objetos almacenados en la base de datos no deberían ser modificados cuando aparecen nuevas vistas. Si se soporta la evolución del esquema, la introducción de una vista OO puede requerir una reestructuración de los objetos almacenados [7], lo cual da lugar a grandes costes por modificaciones en la base de datos, limitando el dinamismo con el que una vista puede ser añadida o eliminada en un sistema relacional. Añadir comportamiento en una vista sí es posible ya que no implica ninguna reestructuración de objetos almacenados.

El problema de la actualización de vistas relacionales aparece también en las vistas OO y está muy relacionado con la identidad de los objetos. Si una instancia de una vista conserva la misma identidad que un determinado objeto almacenado de cierta

clase, entonces cambiar el valor de un atributo de la vista se traduce en una actualización para ese objeto almacenado. En nuestro modelo, las instancias de una vista conservan la identidad de los objetos almacenados de la clase base. Si no se conserva la identidad, algunos tipos de actualizaciones no son posibles. Nuestro concepto de vista OO es aplicable a consultas con operaciones *join* que afectan a varias clases. Una vista puede implicar varios tipos, pero las instancias de la vista tomarán siempre la identidad de uno y sólo uno de los objetos involucrados. Por ejemplo, la vista *Autor-DeExito* implica los tipos *Autor* y *Libro* (un autor de éxito es aquél que ha vendido más de diez mil copias de alguno de sus libros), pero sus instancias toman la identidad de objetos *Autor*. En algunos trabajos sobre vistas OO ya se ha justificado el interés de conservar la identidad de los objetos [9].

Del mismo modo que el estándar ODMG incluye una traducción de los conceptos que define a los lenguajes OO más extendidos (C++, Smalltalk y Java), las vistas deben poder ser trasladadas en términos de estos lenguajes a librerías de clases que implementen exactamente la misma la funcionalidad que define el modelo. Esta traducción es la más clara justificación práctica de la viabilidad de la propuesta.

3 Introducción de Vistas en ODMG

La inclusión de vistas en ODMG ha exigido extender el modelo de objetos, el lenguaje de especificación de esquemas ODL y las ligaduras con C++, Smalltalk y Java. Cabe mencionar que el lenguaje de consultas OQL de ODMG no se ve afectado por la inclusión de vistas, al tener éstas la misma naturaleza que el resto de tipos. A continuación describimos cómo se ha extendido el modelo de objetos y el ODL, dejando para el siguiente apartado el estudio de la traducción a C++ del modelo de vistas.

3.1 Las Vistas Son un Nuevo Tipo ODMG

ODMG distingue tres modos de definición de tipos: interfaces (que definen el comportamiento abstracto de un tipo de objetos), clases (que definen el comportamiento y el estado abstracto) y literales (que definen sólo el estado abstracto). Proponemos introducir las vistas como un nuevo modo de definición de tipos. Por ejemplo, el esquema de base de datos de una empresa, que almacena información sobre empleados y clientes, podría incluir los siguientes tipos:

```
interface Persona {...};
class Empleado : Persona {...};                    // la clase implementa la interfaz Persona
view EmpleadoAntiguo ISVIEW Empleado {...};                  // una vista sobre Empleado
```

La definición de una vista, como *EmpleadoAntiguo*, debe contener lo siguiente:

- Especificación de una clase base, en este caso *Empleado*, junto las propiedades de la misma que serán accesibles en la vista.
- Especificación abstracta del nuevo comportamiento que puede añadir la vista: métodos y atributos calculados.
- Especificación abstracta de la condición que debe cumplir una instancia de la clase base para ser instancia de la vista. Nosotros llamaremos a esta condición *invariante*

de la vista, por ser un concepto similar al de invariante de una clase. No hablamos de la consulta (*query*) sobre la vista, ya que una consulta se asocia a una colección en vez de a una clase.

Igual que las interfaces, las vistas son tipos no instanciables directamente. Una instancia de una interface es siempre una instancia directa de alguna de las clases que la implementan. Lo mismo ocurre con las vistas, ya que los objetos almacenados en la base de datos son instancias directas de clases, que no se ven afectados por la introducción de nuevas vistas.

La relación entre vistas y clases no puede modelarse mediante la relación *ISA*: la extensión (conjunto de posibles instancias) de la vista es un subconjunto de la extensión de su clase base, pero la vista no hereda todas las propiedades de la clase base. El modelado de vistas como subclases da lugar a problemas como la necesidad de clasificación múltiple y de reestructuración dinámica de objetos almacenados [5,7]. Por ello, introducimos una relación de derivación *ISVIEW*, cuya semántica trataremos a continuación.

3.2 Definición de Vistas

Una definición de una vista es una especificación que define el comportamiento abstracto de un tipo de objetos, cuya extensión es un subconjunto de la extensión de algún otro tipo: clase, interface o vista. El siguiente código ODL muestra un ejemplo de declaración de vista:

```
class Empleado : Persona
(  extent empleados)
{
   attribute string nombre;
   attribute float sueldo;
   attribute short anosEmp;
   attribute Empleado jefe;
   void subirSueldo(in float cantidad);
};
```

```
view EmpleadoAntiguo ISVIEW Empleado
{
    invariant esAntiguo;
    attribute string nombre;
    attribute short anoEmpieza;
    attribute EmpleadoAntiguo jefeAntiguo;
    void jubilar();
    void subirSueldo(in float cantidad);
};
```

Como ya hemos comentado, una vista es siempre declarada como vista de un y sólo un tipo, que llamaremos tipo base. En el ejemplo, *EmpleadoAntiguo* es una vista de la clase *Empleado*, y sus instancias toman la identidad de las instancias de *Empleado*.

La definición de la vista puede incluir atributos, asociaciones o métodos definidos en el tipo de base, ya sea directamente o heredados de algún supertipo. Pero la relación *ISVIEW* no es una relación de herencia, por lo que sólo esas propiedades del tipo base serán accesibles en la vista. *EmpleadoAntiguo* no incorpora las propiedades *sueldo*, *anosEmp* o *jefe* de *Empleado*, ni otras propiedades del supertipo *Persona*.

La vista puede añadir otras propiedades, siempre que sean calculadas. Por ejemplo, puede añadir nuevas operaciones, como *jubilar()*. También puede añadir atributos nuevos como *anoEmpieza* o *jefeAntiguo*, pero estos deberán ser siempre atributos calculados, para los cuales se sepa cómo obtener su valor y cómo actualizarlo.

Dentro de una vista se añade un nuevo componente: la especificación abstracta del *invariante de la vista*. El invariante se especifica mediante la cláusula *invariant* seguida del nombre de un método que implementará la condición de pertenencia a la vista; es equivalente a una consulta OQL, pero está definido como un método que

devuelve un valor booleano. Por ejemplo, el método *esAntiguo* podría verificar si el empleado lleva más de diez años en la empresa.

En la declaración de una vista también se puede usar la relación *ISA* para declarar la vista como un subtipo de otro tipo existente. El supertipo de la vista debe ser también un supertipo de la clase base o una vista de alguno de estos supertipos. Por ejemplo, *EmpleadoAntiguo* podría ser declarada como subtipo de *Persona*, o de una vista *PersonaAntigua* (suponiendo que existiera). Al usar esta declaración, la vista hereda todas las propiedades del supertipo y, necesariamente, incorpora el invariante del mismo.

En resumen, podemos destacar las siguientes propiedades del concepto de vista que proponemos. Sea *VA* una vista cuyo tipo base es *A*, entonces:

- La extensión de *VA* es un subconjunto de la extensión de *A*. Son instancias de *VA* las instancias de *A* para las cuales la operación definida como invariante de la vista es cierta. Obviamente, puesto que las instancias de las subclases de *A* pertenecen a la extensión de *A*, también serán instancias de *VA* si cumplen el invariante.
- Las propiedades de *VA* son las declaradas dentro de su definición, que podrán ser propiedades de *A* o bien nuevos métodos o atributos calculados. *VA* no hereda otras propiedades de *A* o de sus superclases implícitamente.
- El tipo *VA* es compatible con *A*, es decir un objeto de tipo *VA* puede ser asignado a una variable de tipo *A*. La asignación de un objeto de tipo *A* a una variable de tipo *VA* requiere una comprobación dinámica del invariante de la vista para el objeto asignado.
- Los tipos *Collection*<*VA*> y *Collection*<*A*> son compatibles entre sí. Siempre se permite la asignación de uno a otro. El significado de esta asignación se verá más detalladamente en el apartado 4.6.

Si, opcionalmente, *VA* es declarado como descendiente de otro tipo *B* entonces:

- *B* debe ser un supertipo (directo o indirecto) de *A*, o una vista de algún supertipo (directo o indirecto) de *A*. En el segundo caso se daría una herencia entre vistas y el invariante de *VA* incluye el invariante de *B*.
- *VA* hereda todas las propiedades de *B*.

3.3 Autorizaciones y Restricciones de Acceso mediante Vistas

El mecanismo de autorizaciones asegura que un usuario sólo usa los tipos a los que haya sido autorizado. Obviamente, el conjunto de tipos autorizados para un usuario debe ser coherente, en el sentido de que las propiedades de esos tipos (los tipos de sus atributos, asociaciones y los resultados de los métodos) deben pertenecer al conjunto de tipos permitidos para ese usuario. Formalmente, se dice que el conjunto de tipos a los que accede un usuario es un cierre transitivo, en el conjunto de todos los tipos definidos, considerando la relación de uso (un tipo tiene un atributo de otro tipo, una relación de asociación con otro tipo o una operación con resultado de otro tipo).

Teniendo en cuenta el principio de coherencia, las propiedades a las que podrá acceder un usuario estarán siempre restringidas a lo que le haya sido autorizado. Por otro lado, la comprobación del invariante de la vista garantiza que una referencia a una instancia de la vista sólo es usada cuando se cumple dicho invariante. Por tanto, la

comprobación de las restricciones se hace en dos partes: en tiempo de compilación (verificar que sólo se usen los tipos permitidos) y en tiempo de ejecución (comprobar los invariantes de las vistas).

4 Traducción de Vistas a Clases C++

Como dijimos anteriormente, es necesario extender la ligadura entre las implementaciones de ODMG y los lenguajes C++, Smalltalk y Java, estableciendo cómo las vistas se traducen en clases del lenguaje. La traducción debe resolver cuestiones tales como: dónde colocar la vista dentro de la jerarquía de clases, cómo hacer que el usuario de la vista sólo acceda a una porción de la información y cómo resolver los problemas de compatibilidad entre tipos. En este apartado se describe la arquitectura del sistema de base de datos OO que permite la traducción de vistas y la traducción a clases C++. El modelo propuesto puede también traducirse a clases Smalltalk y Java.

4.1 Arquitectura del Sistema de Base de Datos

La Fig. 1 muestra una visión global de la arquitectura de un sistema de base de datos que incluya el modelo de vistas propuesto.

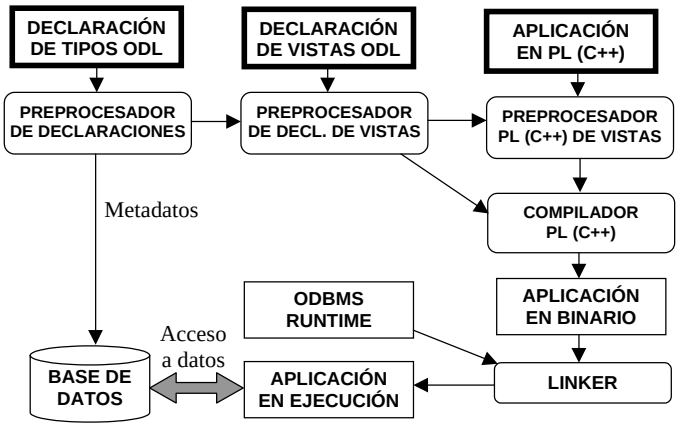


Fig. 1. Arquitectura de un SGBD ODMG incluyendo vistas.

El programador escribe declaraciones ODL para especificar el esquema de la base de datos y una aplicación en algún lenguaje de alto nivel (supondremos C++). Distinguimos entre las declaraciones de los tipos ODL estándar (clases e interfaces) y la declaración de vistas. La definición de vistas es posterior a la de los tipos ODL estándar y, por lo tanto, la declaración de clases e interfaces es independiente del conjunto de vistas definido. El esquema conceptual de la BD no se modifica al variar el esquema externo.

El esquema de la Fig. 1 contiene dos elementos relacionados con las vistas: un preprocesador de declaraciones de vistas y un preprocesador del código fuente de las

aplicaciones. El primero se encarga de traducir las especificaciones de vistas en definiciones de clases C++, que deberán ser implementadas después, en la aplicación C++ o con una librería de clases creadas por el diseñador/implementador de la base de datos. Para hacer la traducción debe conocer el esquema de tipos ODL, para lo cual recibe la salida del preprocesador de declaraciones de clases e interfaces. Este preprocesador definido en el estándar también debe ser modificado, para realizar la conversión a clases C++ de forma ligeramente distinta. El resultado es un conjunto de ficheros de cabecera, para ser incluidos en la aplicación C++.

Pero la traducción de declaraciones de vistas no es suficiente en sí misma y es necesario también un preprocesamiento del código fuente de la aplicación. Este paso asegura que una aplicación sólo usa los tipos a los que esté autorizado. Por último, el compilador recibe como entrada librerías de clases y ficheros de cabecera, con la estructura de tipos definidos en las declaraciones ODL y clases con la funcionalidad ODMG que necesita la aplicación.

4.2 Conversión de Clases ODL

El estándar ODMG establece una equivalencia sintáctica entre los atributos de un tipo ODL y los datos miembros de una clase C++, de forma que un atributo de una clase ODL es traducido en un dato miembro público de la clase C++ correspondiente. Nosotros proponemos introducir aquí una modificación sobre el estándar. Todos los datos serán declarados como protegidos dentro de las clases C++, y la consulta o modificación se realizará a través de los métodos públicos correspondientes. Por ejemplo, sean dos clases *R* y *A*, con las declaraciones ODL mostradas abajo a la izquierda, las clases C++ correspondientes son las que aparecen a la derecha.

class R
{
  readonly attribute long q;
};
	class R : public d_Object {
	 public:
	  virtual d_Long q (void);
	 protected:
	  d_Long _q;};

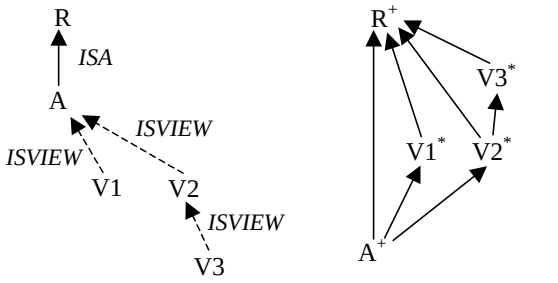
class A extends R
{
  attribute short p;
};
	class A : public virtual R {
	 public:
	  virtual void p (d_Short valor);
	  virtual d_Short p (void);
	 protected:
	  d_Short _p;};

Este cambio está motivado por la necesidad de permitir una modificación del estatus de exportación de un atributo en C++. Una clase *A* puede ocultar una propiedad de una superclase y alguna subclase de *A* puede volver a hacer pública esa propiedad. C++ no permite hacer lo anterior con datos miembros, pero sí con funciones. Esta es la razón de declarar todos los métodos como virtuales. Además, puesto que se va a usar herencia múltiple con compartición, se utiliza herencia virtual. Para mejorar la eficiencia, el compilador puede detectar invocaciones sobre métodos públicos que tan solo retornan el valor de un atributo y tratarlos como acceso directo al atributo.

4.3 Conversión de Vistas ODL

La idea clave de nuestra propuesta es traducir una vista *VA* de un tipo *A*, en una clase C++ del mismo nombre, que es una nueva superclase de la clase *A* y subclase de las clases que corresponden a los supertipos de *A* en la jerarquía de tipos ODL. Si son varios los tipos (clases o interfaces) que *A* extiende o implementa en la jerarquía ODL, *VA* utilizará herencia múltiple para heredar de los mismos. Es decir, *VA* es colocada en la jerarquía entre *A* y sus supertipos.

La clase *VA*, correspondiente a la vista, es una clase abstracta o diferida, que contiene métodos implementados en las subclases y, por lo tanto, no puede ser instanciada directamente. Esto es precisamente lo que buscamos: las instancias de *VA* serán instancias directas de *A* o de sus subclases, las cuales son los objetos almacenados en la base de datos. La Fig. 2 muestra una representación gráfica de un ejemplo de traducción de una declaración ODL en clases C++. El símbolo "*" indica que la clase es abstracta y "+" que es una clase efectiva.



a) Jerarquía de tipos ODL b) Jerarquía de clases C++

Fig. 2. Jerarquía de tipos ODL y su conversión a C++.

Si tenemos la clase *A* definida en el apartado 4.2, y añadimos la vista *V1* como aparece en el esquema de la Fig. 2, entonces la definición de *A* debe modificarse para incluir la vista como un supertipo:

```
class A : public virtual R, public V1 { ... };
```

Recordemos que *A* tiene la propiedad *q* heredada de *R*, y la propiedad *p* no heredada. Entonces, dependiendo de las propiedades que incorpore la vista *V1*, tenemos las siguientes posibilidades:

- ***V1* incorpora la propiedad *p* de *A***
  La clase *V1* debe declarar como públicos y diferidos los métodos correspondientes a esa propiedad. La subclase *A* los hace efectivos, sin necesidad de otros cambios.

```
class V1 : public virtual R {
   public:
      virtual void    p (d_Short valor)=0;
      virtual d_Short p (void)=0;
   ...};
```

- ***V1* no incorpora la propiedad *p* de *A***
  Simplemente, la clase *V1* no debe incluir ningún método relativo a *p*.

- ***V1* incorpora la propiedad *q* de *R***
  La clase *V1* ya hereda esa propiedad de *R* por lo que, igual que antes, no necesita incluir ningún método relativo a *q*. Si otra vista de *A* también incluyera esa propiedad, habría compartición en *A*.

- ***V1* no incorpora la propiedad *q* de *R***
  Los métodos *q* deben ser redeclarados como protegidos, para ocultarlos a los clientes, dándole una implementación vacía. Puesto que *A* hereda esos métodos como protegidos, debe redeclararlos para hacerlos públicos.

```
class V1 : public virtual R {
   protected:
      virtual d_Long q (void);         // no hace nada
   ...};
class A : public virtual R, public V1 {
   public:
      virtual d_Long q (void);         // retornará R::q()
   ...};
```

- ***V1* incorpora una nueva propiedad *t***
  La clase *V1* declara esa nueva propiedad, que deberá ser implementada después. Queda claro que una vista no puede añadir ningún atributo no calculado, y por lo tanto todas las propiedades nuevas serán métodos. De esta forma, las instancias almacenadas de *A* no se modifican. Pero la clase *A* hereda el nuevo método, por lo que lo debe ocultar a sus clientes y subclases. Notar que el método no es declarado como virtual.

```
class V1 : public virtual R {
   public:
      d_Object t (d_String x);
   ...};
class A : public virtual R, public V1 {
   private:
      d_Object t (d_String x) {};
   ...};
```

Si suponemos que existe otra vista *V2* de la clase *A*, se aplicarán las mismas reglas para la traducción a una clase C++. No existen problemas en ninguno de los casos anteriores, ya que *A* comparte todos los métodos heredados que tengan el mismo nombre. Se puede comprobar fácilmente que no ocurre ninguna situación de ambigüedad. Por otro lado, las reglas pueden ser aplicadas independientemente de que el tipo base de la vista sea una clase, interface o vista en la declaración ODL. En el ejemplo anterior, *V3* es una vista de la vista *V2*. Entonces *V3* es añadido como un nuevo supertipo de *V2* y como un subtipo de *R*. A efectos de la conversión, los supertipos de una vista son los de su clase base, pero no la clase base en sí.

En el modelo introdujimos una restricción para la relación *ISA* en el caso de una vista: si una vista *V* es declarada como vista de un tipo *A* y descendiente de otro tipo *B*, entonces *B* debe ser un supertipo (directo o indirecto) de *A* o una vista de algún supertipo de *A*. Por ejemplo, la vista *V3* puede ser declarada como descendiente de *R*, pero no de *V1*, *V2* ó *A*. Esta condición garantiza que no se creará ningún ciclo en el grafo de herencia al hacer la conversión. Además, en general siempre que se utilice esta relación *ISA* dentro de una vista ya se dará esa relación en las clases C++ resultado de la traducción. Por lo tanto, la única implicación de introducir la declaración de *V* como subtipo de *B* es que todas las propiedades de *B* deberán ser accesibles de manera implícita en *V*, es decir, no se deben ocultar.

4.4 El Invariante de la Vista

La extensión de una vista es un subconjunto de la extensión de un tipo, es decir sólo serán instancias de la vista aquellos objetos que cumplan el invariante definido en la vista. Nuestra manera de introducir las vistas en la jerarquía de clases contradice en cierto sentido esta visión, puesto que el tipo base es traducido a una subclase de la vista. El concepto de extensión de una clase, como el conjunto de instancias directas de esa clase y de sus subclases, no modela adecuadamente el concepto de extensión de una vista con la traducción realizada. Por tanto, las instancias de una vista también deberán cumplir el invariante de la misma. Además, puesto que la vista es una clase diferida, todas sus instancias lo serán también del tipo base. De esta forma, la traducción es coherente, ya que una entidad cuyo tipo sea un vista, sólo se podrá conectar a una instancia de su tipo base y un cliente de este tipo no verá propiedades definidas en la vista, aunque la vista sea superclase del tipo base.

El invariante de una vista es un método que no recibe ningún parámetro y devuelve un valor booleano, indicando si el objeto receptor es o no una instancia de la vista. Por ejemplo, una implementación del invariante de la vista *EmpleadoAntiguo*, introducida en el apartado 3.2, podría ser lo siguiente:

```
boolean EmpleadoAntiguo::esAntiguo (void)
{  return this->anosEmp > 10;}
```

Esta propiedad, definida en la clase C++ correspondiente a la vista, está también accesible en la clase base, para permitir su comprobación a los clientes.

Supongamos la estructura de tipos del apartado 4.3. Las vistas *V1*, *V2* y *V3* tienen una operación que actúa como invariante, que llamaremos *v1*, *v2* y *v3*. Por otro lado, todas las clases de la jerarquía C++ tendrán un método *invariant*, que ejecuta el invariante de la clase correspondiente. La definición ODL de la vista *V1* y su traducción a una clase C++ podrían ser las siguientes:

```
view V1 ISVIEW A
{
  invariant v1;
  attribute short p;
  attribute long q;
};
```

```
class V1 : public virtual R {
  public:
    virtual void    p (d_Short valor)=0;
    virtual d_Short p (void)=0;
    virtual boolean v1(void);
    boolean invariant (void){return v1();};
};
```

La clase *A* hereda la operación *v1* y su método *invariant* retorna *true*. Obviamente, el método *invariant* de una clase que corresponde a una clase o interface ODL siempre devolverá el valor *true*.

Puesto que *invariant* es un método no virtual, la versión que se ejecuta depende del tipo estático de la referencia sobre la que se ejecuta (es decir, no hay ligadura dinámica). Si tenemos una instancia de *A*, referenciada a través de las variables *oa* y *ov1*, de tipos *A* y *V1* respectivamente, el mensaje *oa->invariant()* devolverá siempre *true*, mientras que *ov1->invariant()* realizará la comprobación de la vista *V1*.

Una vista *V3* cuyo tipo base es otra vista *V2*, incluirá necesariamente su invariante:

```
virtual boolean v3 (void);
boolean invariant (void) { return V2::invariant() && v3(); };
```

Lo mismo ocurre si la vista es declarada como descendiente de otra vista. Si se dan ambas relaciones, se concatenarán lógicamente con un *and*.

4.5 Comprobación de las Restricciones de Acceso

Como ya hemos comentado, existen dos tipos de comprobaciones que deben ser realizadas para garantizar que un usuario sólo acceda a la información a la que se le ha dado permiso: primero, que la aplicación del usuario sólo use las clases correspondientes a los tipos ODL (clases, interfaces o vistas) que tenga autorizados y, segundo, hacer que la referencia a una vista sólo sea válida (y por tanto usada) cuando se cumple el invariante de esa vista.

En cuanto a lo primero, el preprocesador C++ de vistas es el encargado de hacer las comprobaciones de tipos en tiempo de compilación. El preprocesador recibe como entrada un conjunto de tipos permitidos y el código fuente de la aplicación. Su función es analizar el código fuente del usuario y verificar que los tipos usados por la aplicación (por ejemplo, para definir variables, declarar subclases o hacer conversiones de tipo explícitas) están siempre dentro del conjunto de tipos válidos.

En cuanto a la segunda comprobación, la conversión de ODMG a C++ define la plantilla *d_Ref<T>* para implementar las referencias entre objetos, sean persistentes o no (*smart references*). Una variable de tipo *d_Ref<T>* puede ser derreferenciada usando el operador * y -> o asignada con el operador =. De acuerdo con el estándar, un *d_Ref* a una clase *C* puede ser asignado a un *d_Ref* de una superclase de *C*. Esta asignación es posible gracias a una clase *d_Ref_Any*, que sirve como paso intermedio para la conversión entre distintas instanciaciones de *d_Ref*.

Si el preprocesador C++ garantiza que las referencias a las vistas serán siempre a través de variables de tipo *d_Ref<T>* (es decir, se añade la comprobación adicional de que no se usarán variables de tipo puntero a vista), entonces la comprobación del invariante de una vista queda aislada, como responsabilidad exclusiva, de la clase *d_Ref*. Esto es, será suficiente con verificar el invariante cuando se inicializa o asigna un valor a una variable del tipo de una vista (el operador de asignación está redefinido en la plantilla *d_Ref*) y cuando se ejecute una derreferencia (los operadores * y -> también están redefinidos en el estándar ODMG). En caso de no cumplirse el invariante de una vista para alguna de las anteriores operaciones, entonces se provoca una excepción de tipo *d_Error_RefInvalid*, puesto que estamos intentando acceder a una información a la que no tenemos permiso. En cualquier caso, el cliente podría evitarla introduciendo la comprobación del invariante antes de ejecutar la instrucción que la provocó.

La comprobación del invariante en la asignación (y en la inicialización) no garantiza que éste se siga cumpliendo después, de manera indefinida. Por esta razón, el invariante debe ser comprobado también cuando la referencia es usada con un operador de derreferencia. Esta comprobación añade una sobrecarga de tiempo a las operaciones con vistas, con un coste proporcional al tiempo que requiera la ejecución del invariante. Es claro que se podrían usar técnicas de optimización para evitar en parte este aumento del tiempo de ejecución.

4.6 Conversión de Colecciones

La introducción del mecanismo de vistas tiene importantes implicaciones sobre el manejo de las colecciones. Para un cliente de la base de datos su acceso a la información será normalmente a través de colecciones. Por ejemplo, para una clase *Empleado* se definiría una colección de empleados, sobre la cual un usuario puede hacer con-

sultas, obtener referencias a otros objetos y luego navegar por la base de datos usando *expresiones de camino*. Este tipo de colecciones y los iteradores correspondientes tienen operaciones que devuelven objetos del tipo asociado, por ejemplo *Empleado*, y por lo tanto sólo pueden ser usados por usuarios de la base de datos que tengan acceso a ese tipo.

Una persona que sólo tenga permisos sobre una vista, por ejemplo *EmpleadoAntiguo*, tendrá acceso al tipo de las colecciones de empleados antiguos y a los iteradores correspondientes. Pero, igual que ocurre con los objetos que no son colecciones, los objetos almacenados que maneja un usuario con permisos sobre *Empleado* y otro que usa la vista *EmpleadoAntiguo* son exactamente los mismos, la única diferencia entre ambos es el tipo de las referencias que usan, es decir el tipo dinámico será siempre el mismo, el de la clase almacenada, y el tipo estático es el autorizado al usuario.

El comportamiento de las operaciones en las clases asociadas a colecciones (*d_Set*, *d_Bag*, *d_List*, *d_Array*, *d_Varray*, *d_Dictionary*) y en la clase de los iteradores (*d_Iterator*) debe ser redefinido para tener en cuenta sólo los objetos que cumplen el invariante del tipo al que se instancia la colección. De esta manera, una colección puede contener muchos elementos, pero un cliente sólo podrá ver (a través de las operaciones del tipo) aquellos objetos que cumplan el invariante correspondiente. Esta modificación sólo implica cambios en la implementación genérica de las plantillas de las colecciones y de los iteradores.

Por otro lado, teniendo en cuenta que el estándar ODMG impone la compatibilidad entre una colección de un tipo cualquiera y una colección de un supertipo del anterior, entonces un usuario podrá usar colecciones que contienen objetos de cierta clase *T* a través de referencias cuyo tipo estático es una colección instanciada a un tipo que es una vista de *T*. Por ejemplo, un usuario que sólo tenga acceso a la clase *EmpleadoAntiguo* podrá obtener una referencia a una colección de instancias de la clase *Empleado*, pero a través de una referencia de tipo colección de instancias de la clase *EmpleadoAntiguo*. La implementación de las operaciones en la clase colección hace que el usuario sólo vea los empleados que cumplan el invariante de ser antiguos.

5 Conclusiones y Trabajo Futuro

Hemos presentado una propuesta para la inclusión de vistas en ODMG, centrándonos en los aspectos que consideramos cruciales. Consideramos vistas con la misma funcionalidad de las vistas relacionales, con la misma naturaleza de las clases, y que no generan instancias. La adición de una vista no exige realizar cambios en la base de objetos almacenados. Hemos mostrado cómo es posible traducir el modelo de vistas a clases C++. La traducción a Smalltalk se puede realizar considerando que cada vista se implementa como una clase con un atributo ligado a un objeto de la clase base. Los métodos de la clase base que incluye la vista, se implementan mediante delegación a dicha clase. La traducción a Java sería igual a la presentada en este trabajo, salvo que al no disponer de herencia múltiple, ésta se implementaría mediante interfaces aplicando el esquema habitual (conocido como *forwarding*) [1].

Nuestro trabajo futuro incluirá completar la traducción del modelo de vistas a C++, así como establecer los *bindings* adecuados a Smalltalk y Java, además de abordar la formalización del modelo de vistas, al estilo de lo realizado por Guerrini et al. en [5].

Referencias

1. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Third Edition. Addison-Wesley (2000)
2. Carey, M.J., DeWitt, D.J.: Of Objects and Databases: A decade of Turmoil. Proc. of the 22nd VLDB Conference, Bombay, India (1996) 1-12
3. Cattell, R.G.G. et al.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann Pub. (2000)
4. Garcia Molina, J.J., Guerrini, G., Bertino, E., Catania, B.: Dimensiones en el diseño de un mecanismo de vistas orientadas a objetos. Actas I JIDBD, A Coruña (1996) 119-129
5. Guerrini, G., Bertino, E., Catania, B., Garcia Molina, J.J.: A Formal Model of Views for Object-Oriented Database Systems. Theory and Practice of Object Systems, Vol. 3(3), (1997) 157-183
6. Kim, W., Kelley, W.: On View Support in Object-Oriented Database Systems. En: Kim, W. (ed.): Modern Database Systems. ACM Press (1995) 108-129
7. Kuno, H.A., Rundensteiner, E.A.: The MultiView OODB View System: Desing and Implementation. Theory and Practice of Object Systems, Vol. 2(3) (1996) 203-225
8. Samos, J.: Definition of External Schemas and Derived Classes in Object Oriented Databases. Tesis Doctoral, UPC, Barcelona (1997)
9. Scholl, M.H., Laasch, C., Tresch, M.: Updatable Views in Object-Oriented Databases. Proc. Second Int. Conf. on Deductive and Object-Oriented Databases. LNCS, No. 566 (1991) 189-207
10. Souza dos Santos, C.: Design and Implementation of Object-Oriented Views. Proc. Sixth Int. Conf. DEXA. LCNS, No. 978 (1995) 91-102