Versionado Total de Esquemas a Través de Reglas ECA

Jesús Manuel Maudes Raedo*

José Manuel Marqués Corral**

Carmen Hernández**

Francisco José García Peñalvo*

* Área de Lenguajes y Sistemas Informáticos

Dpto. Ingeniería Electromecánica y Civil

Escuela Universitaria Politécnica, Universidad de Burgos

Avenida Gral. Vigón s/n, 09006 Burgos

e-mail {jmaudes,fgarcia}@cid.cid.ubu.es

Tfno. +34 47 25 89 89, +34 47 25 89 70

Fax +34 47 25 89 56

**Dpto. Informática

Escuela Universitaria Politécnica, Universidad de Valladolid

C/ Fco. Mendizaval 1, 47014, Valladolid

Tfno. +34 83 42 35 02

Fax +34 83 42 31 61

e-mail {jmmc,carmen}@dcs.eup.uva.es

Resumen:

La evolución de esquemas provoca problemas de compatibilidad con las aplicaciones basadas en esquemas definidos en Bases de Datos Orientadas al Objeto (BDOO). La resolución de este tipo de problemas se puede conseguir a base de hacer a los objetos de la base de datos visibles y modificables en distintas versiones del esquema. En esta comunicación se propone la solución al problema de la compatibilidad de aplicaciones y versiones de esquemas mediante la definición de la traducción de operaciones sobre objetos de una versión del esquema en operaciones sobre otra versión diferente del esquema. Esta propuesta para ello amplía la semántica propia de la relación de derivación entre versiones, que habitualmente se limita a guardar los cambios de una versión a otra, con un conjunto de reglas Evento Condición Acción (ECA), donde se definen las mencionadas traducciones de operaciones.

El resultado es una técnica que permite conservar el código de las clases y de las aplicaciones sin necesidad de codificar en unas y otras la adaptación a las estructuras y comportamientos de otros esquemas, respetando el encapsulamiento en las mismas y facilitando las tareas de mantenimiento.

1. Introducción

El grado de utilidad de un sistema de información (SI) se mide, entre otras cosas, por su capacidad de evolución y adaptación a los constantes cambios que se producen en el entorno real que modela. Como consecuencia de este dinamismo de los SI, es necesario efectuar cambios en los esquemas de las bases de datos en explotación que reflejen los cambios de requisitos. Estos cambios dan lugar a nuevas versiones de un esquema que pueden dejar inoperantes aplicaciones que trabajan contra versiones anteriores de dicho esquema, con los correspondientes costes asociados.

No todo gestor de bases de datos permite tener el mismo grado de libertad en las actuaciones de adaptación

que se realizan sobre un esquema; así [Roddick 95] distingue entre cambio, evolución y versionado de esquema. El primero ofrece el soporte habitual de cambios en una base de datos poblada pero con pérdida de información, el segundo garantizaría la no pérdida de la información y el tercero permitiría acceder a la información vieja bajo esquemas recientes, y a la información nueva bajo esquemas antiguos. Si se limita este acceso de tal forma que no se puede hacer modificaciones en los datos de otra versión del esquema, se dice que el versionado es parcial, en caso contrario se dice que es total. Por tanto para resolver el problema de compatibilidad entre versiones de esquema y aplicaciones se debe de desarrollar una aproximación que soporte versionado total de esquemas, la cual será elaborada desde la perspectiva del control de versiones y de la evolución de esquemas.

El control de versiones toma sus orígenes en las bases de datos de diseños CAD y se encarga de la problemática del tratamiento de diseños compuestos de elementos de distintas versiones [Katz 90]. La definición del esquema reside en el diccionario de la base de datos, que a su vez es una base de datos de diseño, que está sometida a diversos refinos en su ciclo de vida, y que finalmente en su mantenimiento sufre evoluciones representables por una secuencia de operaciones que han de ser sometidas a una serie de invariantes que chequeen su consistencia [Banerjee *et Al.* 87], [Breitl *et Al.* 89].

El problema del versionado de esquemas es puesto de manifiesto en [Skarra & Zdonik 87] donde aparece una taxonomía inicial de estos problemas, proponiéndose una solución que recurre a manejadores para mantener la consistencia de comportamiento de las instancias persistentes respecto de las aplicaciones que las utilizan. El problema de este mecanismo es que de alguna manera rompe el encapsulamiento de tipos y clases.

Otro mecanismo de soporte del versionado surge de simular la evolución de esquema a través de vistas [Bertino 92], [Byeon & McLeod 93], [Kim & Kelley 95], o por cualquier otro tipo de mecanismo orientado a la proyección de determinadas partes del esquema, como pudieran ser los objetos multiestructurales de [Tsuda *et Al.* 91]. Este tipo de mecanismos usualmente están orientados a simular el comportamiento de una evolución de esquema para luego probarla antes de llevarla a cabo, no para dar un soporte real al versionado. Las vistas se construyen a partir de clases, de tal forma que obtienen o derivan información ya existente, por lo que no pueden ser utilizadas por ejemplo, para simular la adición de una clase nueva sobre un esquema.

El versionado total se hace necesario frente a la evolución del esquema y la posterior readaptación de las aplicaciones existentes al nuevo esquema, porque el costo que conlleva llevar a cabo esta última solución de una manera rápida es elevado cuando no impracticable. Por ello en el presente trabajo se pretende abrir una vía de exploración en el diseño de mecanismos que sirvan de soporte al versionado total en la que no se rompa el encapsulamiento ni la homogeneidad con el resto del modelo de datos, y de la que finalmente se puedan extraer unas conclusiones que deriven en posteriores trabajos en una implementación real.

2. Relaciones semánticas y servicios propios en el control de versiones

Los servicios del control de versiones son el conjunto de operaciones que el sistema aportará de una forma integrada, que permiten dar soporte al control de versiones. Una descripción de las relaciones semánticas y de los servicios propios del control de versiones se puede encontrar en [Katz 90] y [Bertino & Martino 93]. De entre los servicios propios del control de versiones en el presente trabajo se abordan los de versión en curso y *check out*, *check in* por ser los que tienen una relación directa con el versionado de esquemas, y de entre las relaciones, se abordará la relación derivación por idénticas consideraciones.

Las operaciones de *check in*, *check out* suelen ser descritas en términos de espacios de trabajo en los que se almacenan las versiones por razones de consistencia. Cada tipo de espacio de trabajo es implementado de la misma manera, la única diferencia estriba en quién tiene derecho a acceder a los contenidos de los espacios de trabajo y con qué propósito, (de lectura o de escritura). Habrá un espacio público para las versiones terminadas y espacios de trabajo privados y semiprivados, para diseñar nuevas versiones sin verificar. La operación de *check out* trasladaría un objeto público a un espacio privado para versionarlo, y la operación de *check in*, se encargaría de devolver una versión verificada al espacio público. La operación de *check out*, puede ser como se señala en [Bernstein & Dayal 94] en modo exclusivo, lo que dejaría bloqueado el objeto hasta el momento que se devolviera al espacio público, o en modo compartido, mediante el cual, tan pronto se ejecuta el *check out* sobre un objeto, se produce una nueva versión del mismo en un espacio de trabajo privado. La realización de operaciones de evolución sobre esquemas implican un *check out* exclusivo y un *check in* donde comprobar la consistencia estructural y de comportamiento de la nueva versión del esquema. En cuanto a la versión en curso, es aquella de la que se puede derivar una nueva versión. La versión en curso no tiene porqué coincidir con la última, por lo que será potestad del administrador decidir cuál es.

La relación de derivación, permitiría obtener una nueva versión de un objeto de diseño a partir de otro anterior, dando lugar a grafos dirigidos acíclicos, (DAGs) en aquellos sistemas que permitan conciliar varios

objetos de diseño para obtener una única versión, (lo que obliga a que existan varias versiones en curso), o árboles en otros modelos más simples. Estos DAGs o árboles de derivación se llamarán historiales. Algunos sistemas guardan el historial a partir del registro de las operaciones incrementales que hay que efectuar en el/los ancestro/s directos, para obtener la versión actual, lo que permite controlar la validez del esquema de forma interactiva a partir de la validez de cada uno de las operaciones delta que el usuario va generando como en [Byeon & McLeod 93].

3. Modelización de relaciones a través de reglas ECA

Una regla ECA es una regla compuesta de evento, condición y acción como su acrónimo indica, que permite la modelización de disparadores y restricciones. En una base de datos activa, donde estuvieran definidas estas reglas se ejecutaría la acción que definen, sobre la instancia que dispara el evento, siempre que se verifique la condición.

La ejecución de la acción lleva asociada un modo de disparo como se explica en [Díaz 95], tal que la acción se puede ejecutar antes, (*before*), o después, (*after*), de la ejecución del método que disparó el evento, puede delegar la ejecución en otro evento,(*delegation*), o puede delegarla solo en el caso de que ocurra una excepción, (*exception*). En [Díaz & Paton 94] se aboga por definir en el metamodelo las relaciones semánticas a través de reglas ECA a fin de dotar de extensibilidad y homogeneidad al modelo de datos orientado a objeto.

La relación semántica de derivación tradicionalmente se limita a expresar cuál es la versión ancestro, cuál es la descendiente, y en todo caso el conjunto de operaciones incrementales necesarias para dar el salto a nivel definición entre ambas versiones. Esta aproximación, que parece suficiente en bases de datos de diseño, no cubre las necesidades del versionado de esquemas, al no expresar cómo se puede acceder o modificar las instancias de una versión del esquema desde otra versión del esquema, por ello se propone extender la semántica de la relación de derivación a través de reglas ECA que cubran dichas necesidades, de tal manera que dichas reglas ECA serían definidas por el administrador de la base de datos.

En [Díaz 95] se pretende caracterizar las relaciones en el metamodelo a través de reglas ECA para luego utilizarlas en el modelo. Sin embargo, en el versionado de esquema no se está tratando de modelizar la relación de derivación en general, sino de modelizar qué peculiaridades a nivel de traducción de operaciones tendría cada derivación en concreto, por lo que las relaciones entre versiones no son objetos del metamodelo, sino del modelo como puedan ser las clases; pero con la restricción de que sus reglas ECA han de ser activables al menos desde las dos versiones de esquema que ponen en relación.

En general una relación entre dos clases estará caracterizada por una tupla del tipo:

$$(RID, CID_1, CID_2, A, ECA)$$

Donde *RID* es el identificador de la relación entre las dos clases, que ha de ser único y que podría ser especificado por el usuario. CID_1 y CID_2, son los identificadores de las clases que se ponen en relación, los cuales se puede asumir que son los nombres de las mismas; *A* es el conjunto de atributos asociados a la relación, y *ECA* es un conjunto de *n* reglas ECA asociadas a la relación, que estarán definidas en dicha relación de la misma manera que una clase tiene definidos sus métodos. Por tanto: $ECA \equiv \{ECA_1, ECA_2, ...ECA_n\}$ donde $ECA_i \equiv (Evento_i, Condición_i, Modo\ de\ Disparo_i, Acción\ Procedural_i)$.

Las reglas ECA se activarían con el paso de mensajes expresados en términos del lenguaje de manejo de datos, al intentar realizar operaciones de recuperación, modificación, inserción, borrado, ejecución de un método de una instancia, asociación de objetos, o eliminación de enlaces entre objetos. Estos mensajes generan un evento, que si coincide con alguno definido en las reglas provoca la comprobación de la condición asociada a dicha regla, la cual, caso de satisfacerse dispara la ejecución de la acción procedural en un instante expresado a través del modo de disparo.

La relación de derivación, se define entre una versión de una definición de una clase, y otra definición de otra versión de otra clase, (de igual o distinto *CID*, ya que puede tratarse de un cambio de nombre de la clase o de una reorganización del esquema con lo que un atributo o un objeto entero de la versión antigua puede haber cambiado de clase y se ha de especificar dónde encontrarlo ahora, como en el ejemplo de la Figura 3). Una relación de derivación se traduce formalmente en una relación expresada en la forma:

$$(RID, (CID_1, V_j), (CID_2, V_k), \Delta, ECA)$$

Es decir se trata de una relación binaria, donde cada uno de los *CID*s componentes son a su vez un par formado por el *CID* de la clase que ponen en relación y su número de versión. Es importante resaltar que la versión *j* ha de ser ancestro inmediato de la *k*. Los atributos, Δ, ahora representan el registro incremental de los cambios de definición efectuados para llegar de la clase *j* del esquema ancestro a la *k* del esquema derivado en términos, por ejemplo, de las operaciones de evolución de esquema que aparecen en [Banerjee *et Al.* 87]. Las reglas ECA en este caso representan las acciones que se desencadenarían para traducir las operaciones sobre las instancias en el esquema de la versión *j* en instancias del esquema de la versión *k* y viceversa, como a continuación se verá de forma más ampliada.

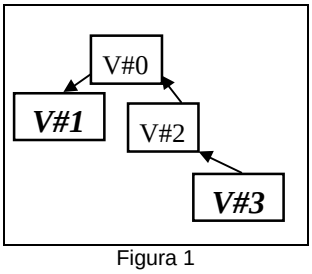

Figura 1

Si se cuida que no exista pérdida de información entre las distintas versiones (*type coercion*), y si se tienen las reglas para traducir una operación sobre una versión del esquema, en una operación sobre otra versión del esquema ancestro o descendiente directo; entonces, se puede traducir cualquier operación sobre una versión de esquema en cualquier operación sobre otra versión del esquema en el historial de versiones debido a la aplicación de la propiedad transitiva sobre la traducción de operadores al ser el historial conexo. Es decir, si el sistema es capaz de definir un camino en el DAG entre las dos versiones, (ver Figura **1**), se puede determinar de manera sencilla cual es el conjunto de disparadores que convierten una operación sobre una versión cualquiera, en otra operación sobre cualquier otra versión.

A continuación se muestran algunos ejemplos de problemas de versionado a los que la aplicación de esta aproximación se adapta perfectamente.

4. Derivaciones a través de cambios en la definición de los atributos y los métodos:

Las operaciones más comunes que pueden afectar a la definición de un atributo son: (1) se elimina un atributo en el esquema actual, con lo cual las aplicaciones antiguas dejarían de funcionar correctamente, (2) se cambia de dominio, por lo que en general ni las aplicaciones antiguas funcionarían con los objetos nuevos, ni las aplicaciones nuevas con los objetos antiguos; (3) se añade un atributo, con lo que las aplicaciones nuevas no funcionarían con los objetos viejos, (4) se ha cambiado de nombre un atributo, (con idénticas consecuencias a (2)).

4.1 Cambio de dominio de un atributo

En el caso de haber cambiado el dominio de un atributo, debería de existir una transformación isomórfica o biyectiva entre ambos dominios para que se pudiera soportar el versionado de esquema. Para ello se definen dos disparadores en la relación de derivación, uno que permita pasar del dominio ancestro al descendiente, y el inverso. La idea de transformación isomórfica parte de una idea originaria de [Kim *et Al* 95], en ese caso aplicada a la integración de esquemas de bases de datos heterogéneas.

Obsérvese que dos transformadores isomórficos definidos sobre dos dominios aplicados consecutivamente sobre un elemento *x* de uno cualquiera de los dominios, deberían de resultar en dicho elemento *x* nuevamente. Esto debe de hacerse así porque de lo contrario se produce el problema de pérdida de información conocido como *type coercion*. Por desgracia esta restricción se hace muchas veces muy fuerte, (típicamente solo será posible para cambios de unidades o de moneda); por lo que en caso de no verificarse se podría intentar ampliar el dominio del atributo a un dominio que sea el dominio unión de los dos anteriores para todo esquema donde el atributo en cuestión sea visible. Esta estrategia garantizaría que ninguna escritura del atributo se pueda producir fuera de dominio, sin embargo habría que controlar que las lecturas de los valores se reconduciesen hacia valores en el dominio de cada una de las versiones.

En [Skarra & Zdonik 87], esta reconducción se codifica a través de manejadores en la propia clase, lo que de alguna manera está rompiendo el encapsulamiento, y enturbiando la codificación de las clases. Por ello se ha optado por cambiar el lugar donde definir estos manejadores, para que estén fuera de las clases. Esto es, en las propias reglas ECA que describen la evolución entre versiones como a continuación se muestra.

Se define un dominio como simple, si una instancia del mismo contiene un único valor, como por ejemplo un entero, un real, una cadena etc.. En contraposición, un dominio estructurado será aquel que se exprese en términos de algún constructor de tipo como son tuplas, conjuntos y listas. Inspirándose en [Abiteboul *et Al* 95], se puede definir formalmente un dominio estructural como:

1. Una tupla, que se denota por $[...,a_i: t_i,...]$, donde a_i es el i-ésimo nombre de atributo de la tupla y t_i es el dominio de dicho atributo, que a su vez puede ser simple o estructurado. Cada atributo tiene un significado producto de una abstracción hecha por su diseñador.

2. Un conjunto de elementos $\{t_i\}$, donde t_i es el dominio de sus elementos, que a su vez puede ser simple o estructurado.

3. Una lista de elementos $<t_i>$, donde t_i es el dominio de sus elementos, que a su vez puede ser simple o estructurado.

Se dice que un dominio *A* está incluido en otro *B*, si:

- *A* y *B* son dominios simples y se cumple que $\forall x \in A \Rightarrow x \in B$.
- O bien, *A* y *B* son dominios estructurales del tipo conjunto, y el dominio de cada elemento de *A* está incluido en el dominio de cada elemento *B.*
- O bien, *A* y *B* son dominios estructurales del tipo lista, y el dominio de cada elemento de *A* está incluido en el dominio de cada elemento *B.*
- O bien, *A* y *B* son dominios estructural del tipo tupla, y para todo A_i atributo de *A* existe un atributo B_j de *B* tal que su significado sea el mismo y el dominio de A_i está incluido en el dominio de B_j.

Se dice que un dominio *A* recubre a un dominio *B,* si el dominio de *B* está incluido en *A*.

Cuando un domino está incluido en otro, como pueda ser *Precio* en el ejemplo de la Figura 2, el dominio más amplio actuará de unión o recubrimiento de ambos. En el ejemplo de la Figura 2. el dominio de los enteros actúa como unión de enteros y un subrango de los enteros. Sin embargo, encontrar un dominio recubrimiento de tipos que no están estrictamente incluidos el uno en el otro o que no sean dominios simples, puede ser una tarea más compleja; (por ejemplo la clase coche que tiene componentes de la clase rueda – coche, que han sido versionados en dos clases estructuradas que no están incluidas la una en la otra, tal y como muestra la Figura 2).

En tal caso lo que se hace es definir una clase de apoyo que sea el recubrimiento o unión de las dos versiones anteriores, (similar a los *version set interface* en [Skarra & Zdonik 87]). En el ejemplo de la Figura 2 hemos definido la versión recubrimiento RuedaCoche#V1_2; esta nueva versión 1_2 de apoyo contendrá todos los atributos de las dos versiones anteriores.

De manera recursiva en la versión recubrimiento, si uno de los dominios está incluido en otro y ambos son dominios simples, los atributos se definen entonces en dominios unión de los dominios de los dominios de los que a su vez procedan. Por ejemplo el atributo *Precio* en la versión 1_2 ha sido tomado como entero por ser este dominio unión de enteros y 1000..75000. En caso contrario no es necesario crear por el momento una clase recubrimiento de estos dominios; bastará con utilizar la que se cree en el momento de definir la relación de derivación entre los dominios de los dos atributos. Así, la clase coche seguirá referenciando a las ruedas independientemente de su versión, siendo la relación de evolución entre las versiones 1 y 2 de rueda, la que define una versión 1_2 recubrimiento de las anteriores.

Al final de la definición de una versión de recubrimiento, siempre aparecería un atributo especial que expresa qué rol está desempeñando cada objeto recubrimiento en cada momento, (bien de versión 1, o bien de versión 2). Esta técnica es similar a los *mensajes estructurales* de [Tsuda *et Al* 91], o a los *slicing objects*, [Kuno *et Al* 96]. El desempeño de un rol por una instancia de la versión 1_2, implicará que tenga el mismo interfaz que una instancia de la versión que interpreta, enmascarando el resto de atributos y métodos. Sea cual sea el rol de una instancia de una clase recubrimiento, seguirá siendo instancia de la clase recubrimiento y no de la clase cuyo rol está interpretando. Un rol es por tanto, una máscara que impide acceder a determinados atributos o métodos de una clase recubrimiento. Una clase que no sea recubrimiento obviamente está representando el rol de ser ella misma. La restricción de acceso a un atributo de una clase recubrimiento enmascarado por un rol, en última instancia, se puede implementar a través de una regla ECA que levante una excepción ante tal evento.

El tratamiento analizado para cambios estructurales en dos clases, se puede extender a versiones de varias clases, con el consiguiente crecimiento de la clase de apoyo, que habrá de recubrir todo el conjunto de versiones de la clase en cuestión.

Obsérvese que el manejo de los roles solamente compete al administrador de la base de datos, ya que es necesario para definir las reglas ECA de una relación de derivación. Los roles son transparentes a la concepción que tiene un usuario no administrador del sistema; de hecho, la creación de la versión de apoyo 1_2 podría dejarse como una tarea automatizable por el sistema o como una tarea propia del administrador.

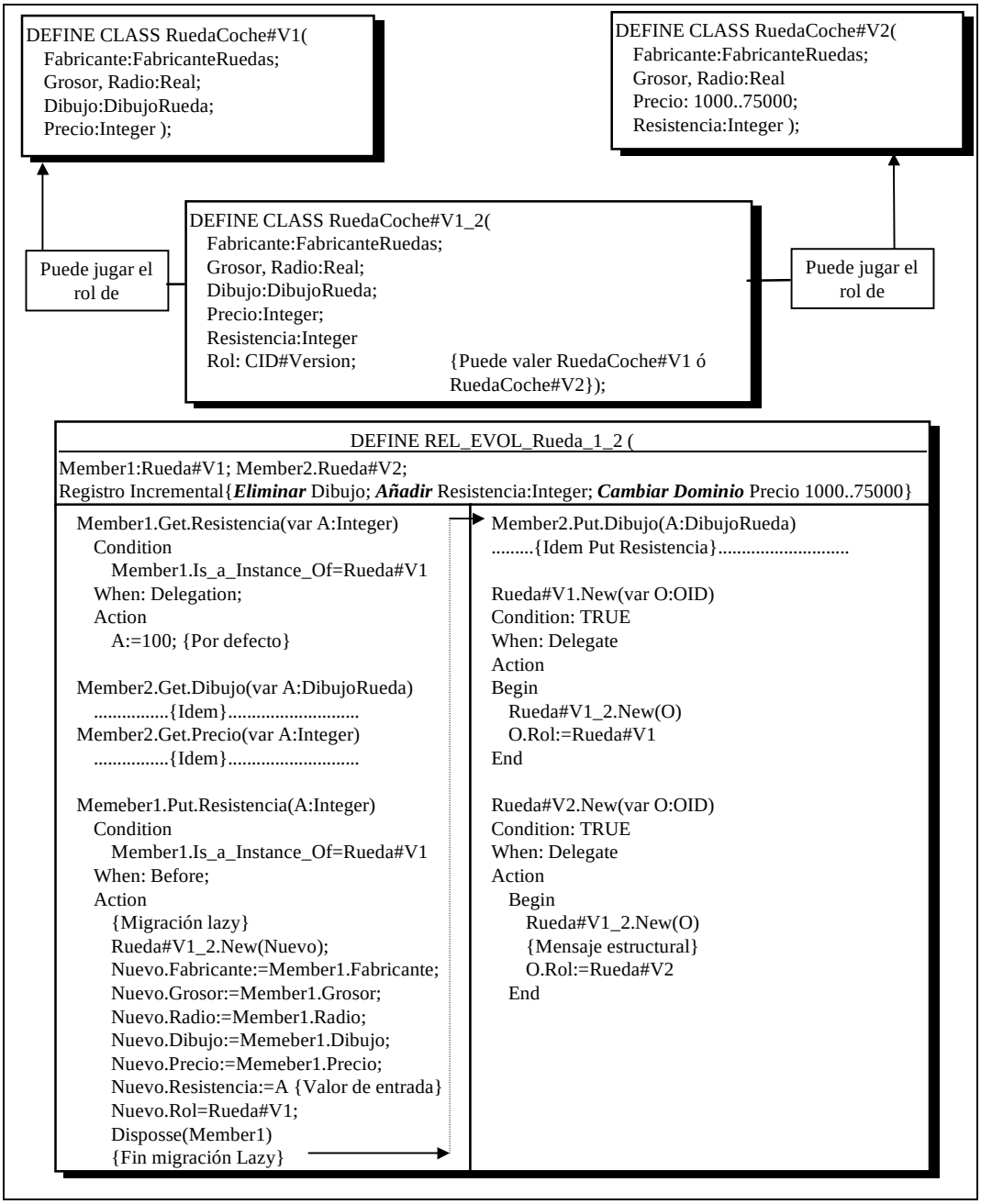

Figura 2.

En el ejemplo de la Figura 2. se utiliza una técnica de migración de poblaciones no estricta o *lazy* al estilo de ORION [Banerjee *et Al* 86],[Banerjee *et Al* 87],[Kim & Chou 88] hacia la clase recubrimiento de las versiones 1 y 2. Los accesos a un atributo añadido devuelven el valor por defecto establecido en la definición de la clase o el valor '*NULL*' si no hubiera existido ningún valor por defecto. Solo se produce la adición del atributo a las instancias en el momento de modificar su valor. Esta estrategia no tendrá que ocasionar por tanto, *overhead* por modificación de esquema como en el caso de las conversiones estrictas o centralizadas que utilizan otros sistemas como *GemStone*, [Penney & Stein 87]. En cualquier caso, lo importante es observar que la estrategia de migración de poblaciones es definible, en última instancia, por el administrador, en las acciones de la reglas ECA, con lo que se puede evitar la utilización de valores por defecto y valores nulos, que frecuentemente desvirtúan la semántica de los atributos.

4.2 Aparición de nuevos atributos, eliminación atributos y renombrado de atributos

La aparición de un nuevo atributo trae como consecuencia que la nueva versión recubra a la anterior, de donde la clase recubrimiento se podría ver como una especialización de la nueva versión de la clase a la que se le ha añadido la capacidad de soportar los roles de ambas versiones. El tratamiento de la eliminación de un

atributo es el simétrico al de la adición de atributos.

El cambio de nombre de un atributo de una clase *A* para transformarla en una clase *A'* no necesita de una clase recubrimiento. Esto es consecuencia de la definición de inclusión de dominios que se ha dado, en virtud de la cual *A* estaría incluido en *A'* y viceversa, por lo que se concluiría que ambos dominios serían el mismo. Es por ello que bastaría definir un par de reglas ECA en la relación de derivación que deleguen las operaciones sobre el atributo renombrado en el correspondiente de la clase de la que el objeto implicado sea instancia.

4.3 Cambios en los métodos

El tratamiento que se ha hecho, hasta ahora, a las versiones de las clases parece haber olvidado que estas tienen métodos. No obstante, este olvido es intencionado, pues un método, al final, no es más que una pequeña aplicación definida para una versión del esquema; como en todo momento se está cuidando que las aplicaciones diseñadas para una versión del esquema sigan funcionando sin tener que tocar la codificación de clases y aplicaciones, se concluye que las operaciones sobre los métodos no requieren de ningún tratamiento especial por parte del administrador.

Por ejemplo: sea una clase *A* con un método *m*, tal que O_1 es instancia de *A*; además *A* ha evolucionado a *A'*, por redefinición del método *m*, que se denotará como *m'*. Si posteriormente el objeto O_2 se instancia bajo el nuevo esquema de *A'*, puede ocurrir que los programas que trabajaban con el esquema de *A*, invoquen al objeto O_2 definido en *A'*. En dicho caso, se debería de ejecutar el método definido en *A* que es *m*; como dicho método tiene una signatura y unos requisitos compatibles con el programa que lo está utilizando no habría en este sentido ningún problema. Se podría pensar que, dentro del método antiguo, se puede utilizar algún atributo que ya no está definido en *A'* o que ha sido redefinido para otro dominio, pero esto no tendría que ser tampoco problema, ya que se solucionó en la sección anterior a través de un disparador que devuelve un valor por defecto o transforma el atributo a un dominio compatible con *A*. El análisis inverso también se podría hacer para programas que trabajen con *A'* y accedan al objeto O_1.

No obstante, si se han suprimido atributos utilizados por el método, pueden ocurrir efectos indeseables al trabajar con las traducciones de operaciones en las instancias nuevas, ya que usualmente devolverán valores por defecto del atributo. Este caso se puede solucionar redefiniendo dicho método en la acción de una regla ECA de la relación entre ambas versiones de las clases, de tal forma que no se utilice, los atributos eliminados, y se ejecute solo en el caso de que la instancia pertenezca a la versión de la clase que no tiene dichos atributos.

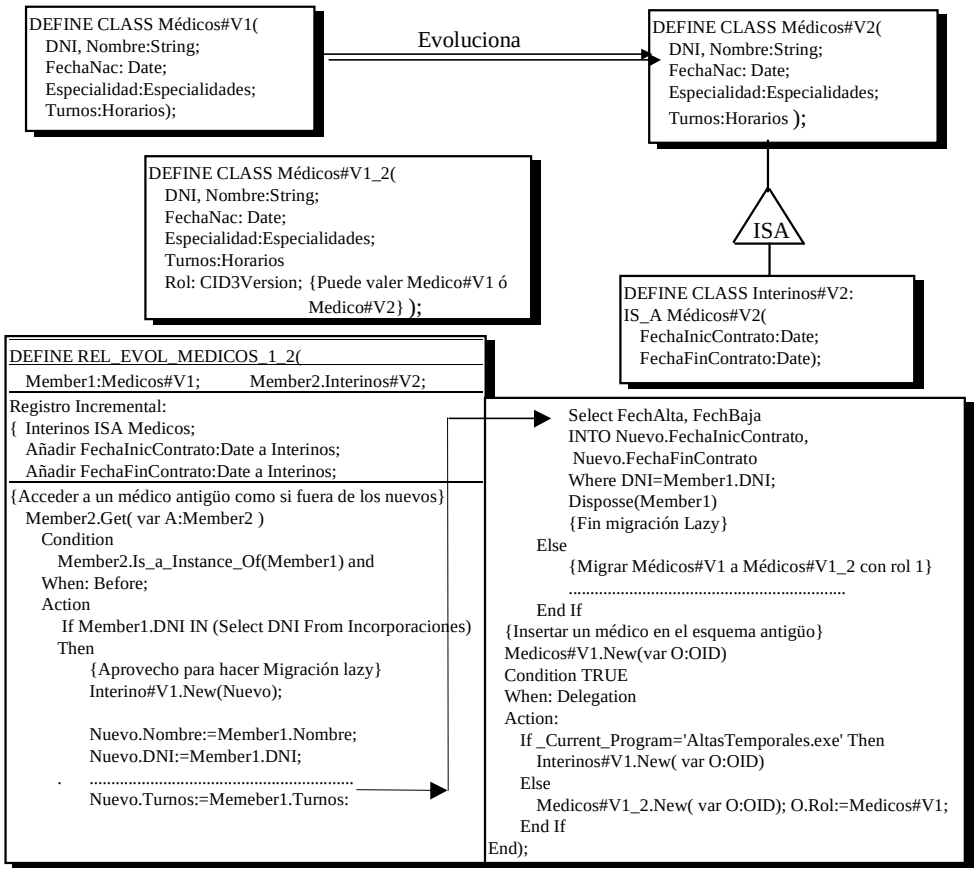

Figura 3

4.4 Creación y eliminación de métodos

La creación o eliminación de métodos tampoco tiene ningún efecto de cara al administrador, puesto que solo serán visibles en las versiones del esquema en que estén definidos, por lo que serán únicamente aquellas aplicaciones diseñadas para esa versión del esquema las que los invoquen.

Como el desempeño de un rol por parte de una instancia de una clase recubrimiento implica que esa clase tiene el mismo interfaz que la clase interpretada por el rol, es imposible el acceso a los métodos que no sean propios de un rol, por parte de las instancias de clases recubrimiento.

Los métodos están definidos tan solo en las clases que no son de recubrimiento, y no en las instancias, lo que evita tener que plantearse una migración de poblaciones hacia clases recubrimiento para hacer posible el acceso a los métodos desde distintas versiones del esquema.

5. Derivaciones a través de reorganizaciones de la jerarquía de herencia

Al hacer una operación de reorganización de la jerarquía, las instancias de una clase puede que se hallan repartido, en varias clases caso de que se haya sufrido una especialización por evolución, o que las instancias de varias clases especialización se hayan reunido en una única clase, caso de que se haya sufrido una generalización por evolución. Tras estas operaciones en distintas versiones del esquema ocurrirá que la misma clase aglutinara de forma directa distintos identificadores de objeto.

Supóngase un primer caso en el que una clase *A* es una especialización de una clase *B*, si entonces se evoluciona a que *A* como especialización desaparezca, todas sus instancias en la siguiente versión aparecerían como instancias de *B*. Si un nuevo objeto se intenta insertar en la nueva versión, cuando el sistema acceda a él mediante la versión anterior, ¿cómo habrá de buscarlo?, ¿como instancia de *A* o de *B*?. Por supuesto, que aunque sea transitivamente, el objeto pertenece a la generalización B; pero no queda claro a qué clase pertenecería de forma directa.

En un segundo caso se tendría una clase *B* y a partir de ella se derivaría una especialización *A*, (Figura 3). El problema surge aquí al intentar insertar un objeto de la versión vieja y no saber si se corresponde o no con la especialización que se acaba de definir.

En ambos supuestos se necesita una regla ECA definida en la relación de derivación que al acceder a un objeto en la versión que mantiene la subclase, decida si es instancia directa de la generalización o de la especialización, migrando la instancia hacia la especialización si fuera el caso.

6. Conclusiones y futuros trabajos

La evolución de los esquemas de bases de datos trae consigo la incompatibilidad de las aplicaciones viejas, poniendo en peligro la capacidad de cambio de los sistemas de información. Por ello se han diseñado una serie de mecanismos que permitirían la coexistencia de múltiples esquemas de bases de datos sobre un mismo conjunto de objetos persistentes.

Estos mecanismos se basan en definir en las relaciones de derivación propias del control de versiones un conjunto de reglas ECA que traducen manipulaciones y accesos de instancias de clases sobre objetos de un esquema a manipulaciones y accesos de instancias de otro esquema. La transitividad de la relación de derivación y la imposición del isomorfismo en las transformaciones, permite extender el modelo de traducción entre dos esquemas a un modelo de traducción entre varios esquemas en general.

Aunque no se han cubierto todas las operaciones que se apuntaban en [Banerjee *et Al.* 87], se han tratado las que probablemente tengan mayor dificultad, como son los problemas con la redefinición de dominios de los atributos, y los problemas con los repartos de instancias por una reorganización de la jerarquía de herencia. Las operaciones que quedan por tratar o no se corresponden con operaciones que se puedan hacer en cualquier modelo (p. ej.: atributos compartidos), o tienen una solución evidente. De entre estos últimos destacan: (1) Los problemas de resolución de la herencia múltiple, cuyos criterios de resolución ahora pueden ser especificados al antojo del administrador en el metanivel, como una regla ECA. (2 y 3) Los problemas relativos a la creación de nuevas clases y eliminación de clases se reducen a la creación o eliminación de todos los atributos y métodos de la clase, problema que ya se ha tratado. (4) Los cambios de nombre de las clases implicarían una solución similar a la del cambio de nombre de un atributo.

Aunque funcionalmente el mecanismo que se ha propuesto es consistente, si parece claro que la acumulación de transformaciones a causa de la coexistencia de versiones de esquemas degenerará en un *overhead* que entorpecerá el funcionamiento óptimo del sistema. Por lo tanto, en el momento que un esquema,

que no es el actual, no es utilizado por ninguna de las aplicaciones antiguas, se debería de tratar de eliminarlo. Si el diccionario del sistema guarda las versiones a través de un registro incremental de las modificaciones, (*deltas*), como es el caso del modelo propuesto, tendrá que existir un proceso que reconstruya las definiciones a partir de la secuencia de cambios, (lo que [ISO 96] llama *materialización del working set*), lo que condiciona la eliminación física de una versión a si existe o no otra versión basada en ella. Por lo que solo se pueden eliminar las hojas y la raíz en el caso de que tuviera un único descendiente; lo que influirá sin duda en el orden de prioridad en el que el equipo de desarrollo debe de migrar las aplicaciones antiguas hacia el esquema nuevo. Caso de hacer una eliminación de una versión habrá que migrar los objetos de la versión eliminada a alguna de las versiones activas, idealmente hacia aquella que favorezca la eficiencia del sistema, la cual heurísticamente podría coincidir con la más utilizada.

La aproximación que se ha desarrollado tiene como ventaja principal que toda lógica asociada a la traducción de operaciones sobre distintas versiones de esquemas no reside en las propias clases, sino que estas se mantienen limpias dejando la codificación en la definición de las relaciones de derivación, dando como fruto un sistema fácil de mantener y de asimilar por el administrador.

Otros campos de aplicación del mecanismo propuesto serían (1) la adaptación al problema de la integración de esquemas en bases de datos distribuidas heterogéneas, donde cada esquema componente exportado al modelo de datos común puede ser visto como una versión del esquema federado; (2) su extensión a modelos de versionado que permitan la reutilización de partes de esquemas definidos en otras versiones que no fueran la versión en curso, lo que daría lugar a la definición en términos de reglas ECA, de referencias dinámicas y mecanismos de propagación de cambios como los que se discuten en [Katz 90].

Estas últimas extensiones expresadas en el punto (2) se estudiarán en la definición de un modelo de repositorio para la reutilización de componentes *software* con soporte para la evolución y extensión del propio esquema del repositorio.

En cuanto a la implementación del modelo que se ha descrito en el presente trabajo para su estudio, se han valorado algunos gestores de bases de datos orientadas al objeto, siendo *Postgres* uno de los que mejor se adapta a causa de sus características activas. Por lo que nuestro trabajo inmediato será implementar el modelo sobre este gestor para, posteriormente generar un asistente del administrador, que como respuesta a las operaciones de evolución de esquema, le sugiera la codificación de las reglas ECA necesarias para llevar a cabo el versionado, en una línea similar a la que aparece en [Byeon & McLeod 93] para generar las vistas que simulan el versionado. También se debe de estudiar la implementación de un lenguaje de manejo de datos que trabaje contra varias versiones de esquemas de forma transparente al usuario.

Referencias

[Abiteboul *et Al.* 95] Abiteboul S., Hull R., Vianu V. *Foundation of Databases*. Adison Wesley Publishing Company. 1995.

[Banerjee *et Al.* 86] Banerjee, J.; Chou, H.T., Kim, H.J., and Korth, H.F. *Schema Evolution in Object Oriented Persistent Databases.* En Proc. 6th Advanced Databases Symposium, Tokyo (1986) pp. 23-31.

[Banerjee *et Al.* 87] Banerjee, J.; Chou, H.T., Kim, H.J., and Korth, H.F. *Semantics and Implementation of Schema Evolution in Object Oriented Databases.* ACM SIGMOD Conference, SIGMOD Record Vol 16 No. 3 (1987) pp. 311-322.

[Bernstein & Dayal 94] Bernstein, P.A., Dayal, U., *An Overview of Repository Technology* En Procs. of the 20th VLDB. Santiago, Chile (1994). pp 705-713.

[Bertino 92] Bertino E., *A view mechanism for object oriented databases*. En Proc. 3rd International Conference on Extending Database Technology (EDTB). Viena, Austria, 1992.

[Bertino & Martino 93] Elisa Bertino, Lorenzo Martino. *Object Oriented Database Systems Concepts and Architectures.* Adison Wesley 1993.

[Breitl *et Al.* 89] Breitl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., and Williams, M. *The GemStone data management system*. En Kim, W. & Lochovsky, F. (eds). Object Oriented Concepts, Databases and Applications. ACM Press (1989) pp 283-308.

[Byeon & McLeod 93] Byeon, K.J., McLeod, D., *Towards the Unification of Views and Versions for Object Databases*. En Object Technologies for Advanced Software; 1st JSSST International Symposium Kanazawa, Japan, Nov 1993. Lecture Notes in Computer Science 742. Springer Verlag. pp 220-235.

[Díaz 95] Díaz O., *The Operational Semantics of User Defined Relationships Oriented Database Systems*. Data & Knowledge Engineering 16 (1995) pp 223-240.

[Díaz & Paton 94] Díaz, O., Paton, N. W., *Extending ODBMSs Using Metaclasses*. IEEE Software May 1994, pp 40 - 48.

[ISO 96] *Interim Draft of Revised IRDS Services Interface*. ISO\IEC UK proposal. 15/05/96.

[Katz 90] Randy H. Katz, *Toward a Unified Framework for Version Modeling in Engineering Databases*. ACM Computing Surveys. Vol. 22, Nº4, December 1990.

[Kim & Chou 88] Kim, W., Chou, H.T., *Versions of Schema for object oriented databases*. En Bancilhon, F. and DeWitt, D. J. (eds) Proc. 14th VLDB, Los Angeles, CA (1988) Morgan-Kaufmann. pp 148-159.

[Kim & Kelley 95] Kim,W., Kelley, W., *On View Support in Object - Oriented Database Systems*. Modern Database Systems: The Object Model, Interoperability, and Beyond; Kim. W. (eds). ACM Press. New York. 1995. pp 108-129.

[Kim *et Al* 95] Kim, W., Choi, Y., Gala, S., Scheevel, M., *On Resolving Schematic Heterogeneity in Multidatabase Systems*. Modern Database Systems: The Object Model, Interoperability, and Beyond; Kim. W. (eds). ACM Press. New York. 1995. pp 521-550.

[Kuno *et Al* 96] H.A. Kuno, E.A. Elke, A. Rudensteiner, *The Multiview OODB View System: Design and Implementation*. En Theoty and Practice Of Object Systems, (eds). John Wiley & Sons, Inc., Vol. 2(3), 1996.

[Penney & Stein 87] Penney, D.J. and Stein, J. *Class modification in the GemStone object oriented DBMS*. SIGPLAN Not. Proc. OOPSLA 87. Vol. 22 No 12 (1987). pp 111-117.

[Roddick 95] John F. Roddick. *A survey of schema versioning issues for database systems.* Inf. Soft. Technology, Volume 37 Number 7, 1995 pp 383-393.

[Skarra & Zdonik 87] Skarra, A.H., Zdonik, S.B. *Type Evolution in a Object-Oriented Database*. Research Foundations In Object Oriented and Semantic Database Systems. Alfonso F. Cárdenas, Dennis Mc Leod Editors. Prentice Hall. pp 137-155.

[Tsuda *et Al.* 91] Tsuda, K., Yamamoto, K., Hirakawa, M., Tanaka, M., Ichikawa, T. *MORE: An Object - Oriented Data Model with a Facility for Changing Object Structures*. IEEE Transactions on Knowledge and Data Engineering, Vol. 3, Nº 4, Dec. 1991. pp 444-460.