Generación Automática de Restricciones de Integridad: Estado del Arte

Jordi Cabot
Estudis d'Informàtica i Multimèdia
Universitat Oberta de Catalunya
Seu Central
08035 Barcelona
jcabot @uoc.edu

Ernest Teniente
Dept. de Llenguatges i Sistemas Informàtics
Universitat Politècnica de Catalunya
Campus Nord
08034 Barcelona
teniente@lsi.upc.edu

Resumen

La aparición de la MDA (Model-Driven Architecture) y, en general, del MDD (Model-Driven Development) ha dado un nuevo impulso a la investigación acerca de la generación automática del código de una aplicación a partir de su especificación. En este artículo se estudia el soporte ofrecido por los métodos y herramientas actuales para la generación automática de las restricciones de integridad definidas durante la especificación. En concreto, se muestra que el soporte ofrecido es todavía bastante limitado, ya sea en cuanto a la expresividad de las restricciones permitidas o a su eficiencia. Con base a esta evaluación se definen las características que tendría que poseer un método de generación ideal.

1. Introducción

La generación automática de código es ya un viejo sueño (ver por ejemplo [1]), al que la MDA [2] (Model-Driven Architecture) y, en general, el MDD (Model-Driven Development) han dado un nuevo impulso.

Últimamente han aparecido muchos métodos y herramientas que *prometen* la generación automática y completa del código de una aplicación a partir de su especificación en UML. Como ejemplo, es difícil encontrar una herramienta CASE que no se anuncie a sus posibles usuarios/compradores destacando sus capacidades de generación de código o su adhesión a la filosofía MDA.

No obstante aún queda mucho por hacer. Todos estos métodos y herramientas son capaces de generar clases en Java o tablas en una base de datos relacional (BDR) a partir de un esquema conceptual (EC) definido usando un diagrama de clases en UML y (algunos menos) de generar también la parte dinámica a partir de diagramas de estados o lenguajes de acciones (Action Semantics, [3]). El problema es que muchos de ellos se "olvidan" de las restricciones de integridad (RI) durante esta generación, a pesar de que, tal y como se define en [4] las RI son una parte fundamental de la especificación de una aplicación y por lo tanto tienen que tenerse en cuenta durante su implementación.

En este artículo se estudia el soporte ofrecido por estos métodos para la generación automática de las RI definidas en la especificación del EC. Como se verá, todos presentan limitaciones en cuanto a la expresividad de las RI permitidas o respecto a la eficiencia del código generado para su comprobación. Además, en muchos de ellos el soporte es casi nulo.

Es totalmente imposible evaluar todas las herramientas y métodos disponibles. Se han intentado escoger los más representativos de cada grupo (herramientas CASE, herramientas MDA, métodos de generación automática…). Se han incluido también todas las herramientas que permiten la definición de RI en OCL (Object Constraint Language [5]) o similares, ya que son las únicas que pueden permitir la máxima expresividad en la definición de RI (la mayoría de RI no se pueden expresar simplemente de forma gráfica y necesitan de un lenguaje específico [6 ch. 2]).

La estructura de este trabajo es la siguiente: en primer lugar se definen los criterios de la evaluación. A continuación, en la sección 3 se evalúan las diferentes herramientas y métodos. En la sección 4 se definen una serie de características deseables en todo método de generación de RI. Finalmente, en la sección 5 se presentan algunas conclusiones.

2. Criterios de evaluación

En esta sección se presentan los criterios seguidos para la selección y/o evaluación de las diferentes herramientas.

2.1. Expresividad en la definición de RI

Algunas RI son inherentes al lenguaje de definición utilizado pero la mayoría necesitan especificarse explícitamente. De éstas, algunas están predefinidas y en general pueden expresarse gráficamente (como las restricciones de cardinalidad). Otras necesitan definirse utilizando un lenguaje específico, comúnmente OCL o derivados.

Dentro de las herramientas que permiten la utilización de un lenguaje específico para la definición de RI también hay diferencias respecto a los operadores permitidos en la definición. Por ejemplo, a veces las RI no pueden contener navegaciones entre objetos y por lo tanto sólo pueden afectar diferentes atributos de un mismo objeto (restricciones a nivel de objeto según la clasificación de [7]). Cuando se permiten navegaciones tenemos restricciones inter-objeto. Finalmente, si permiten comparaciones entre diferentes objetos de una misma clase tenemos restricciones a nivel de clase (sería el caso, por ejemplo, de las restricciones que contienen el operador *allInstances* de OCL). Dentro de cada grupo también se debe tener en cuenta si permiten operadores de agregación, negaciones, operadores conjuntivos…

Muchas veces, aunque la herramienta no permita ciertos tipos de operadores en la definición de RI, sí que ofrece tipos de RI predefinidos que los utilizan. Por ejemplo, puede no dejar utilizar operadores de navegación en las RI pero si ofrecer restricciones de cardinalidad (que implican una navegación entre un objeto y los objetos relacionados de otra clase) o de unicidad (que implican la comparación del valor de un atributo con los valores de todos los demás objetos de la clase para ese mismo atributo). La mayoría de veces es debido a que la tecnología final para la que la herramienta es capaz de generar código incluye una construcción específica para ese tipo de RI predefinida. Por lo tanto la herramienta no tiene que preocuparse de generar el código que comprueba la restricción sino simplemente traducir la definición de la RI al lenguaje correspondiente en la tecnología final.

2.2. Eficiencia del código generado

Una RI define una condición que la Base de Información (BI) debe satisfacer siempre. Como consecuencia, después de cada modificación de la BI (debida a la ejecución de una operación o transacción) el código generado tiene que comprobar que la BI sigue verificando la condición.

El primer nivel de eficiencia radica en comprobar la RI sólo después de cambios en la BI que puedan violar la RI. Por ejemplo, si el esquema conceptual contiene una RI que obliga a todos los empleados a ser mayores de 16 años, no es necesario comprobar la RI cuando se eliminan empleados o se crean instancias de otras clases diferentes de la clase empleado.

El segundo nivel consiste en verificar la restricción usando el menor número de objetos posible (comprobación *incremental*). Siguiendo el ejemplo anterior, una vez damos de alta un nuevo empleado sólo es necesario comprobar la edad del nuevo empleado pero no las edades de todos los demás. Hay diferentes grados de incrementalidad, ya que dependiendo de cada método se consigue evaluar determinados tipos de restricciones usando más o menos objetos.

En el presente artículo sólo vamos a evaluar la eficiencia obtenida por el código directamente generado por la herramienta a partir del esquema conceptual. Para determinadas tecnologías (como bases de datos deductivas) existen ya métodos específicos (ver [8]) capaces de procesar una RI para conseguir su evaluación incremental. Por lo tanto, en este caso una opción seria transformar la RI a lógica o SQL (operación no trivial) y aplicar uno de estos métodos para conseguir la eficiencia. Evidentemente esta opción sólo es posible para tecnologías para las cuáles existen estos métodos (básicamente bases de datos deductivas y relacionales).

2.3. Tecnologías soportadas

La base de información de la aplicación sobre la que queremos verificar las restricciones estará implementada usando alguna tecnología en concreto.

Lo más habitual es usar como base de información una base de datos relacional. En este caso las restricciones se comprueban sobre la información guardada en forma de tuplas en la base de datos. Otra opción es mantener una representación de la base de información en memoria principal mediante la definición de un conjunto de clases creadas usando un lenguaje orientado a objetos. En este caso los objetos normalmente acaban igualmente haciéndose persistentes en una base de datos relacional aunque las RI no se comprueban sobre los objetos en la base de datos sino sobre los objetos en memoria interna. En otros casos, como en las aplicaciones de tiempo real, los objetos pueden no almacenarse nunca en memoria secundaria o utilizar soportes menos sofisticados.

Por lo tanto, a la hora de estudiar las capacidades de generación de RI se tendrán en cuenta estas dos tecnologías: 1 – Base de datos relacional, 2 – Lenguaje orientado a objetos, en particular el lenguaje Java.

Aunque muchas de las herramientas a estudiar son capaces de generar código para otras tecnologías (como .NET o C++), esta elección no restringe las herramientas a considerar ya que estas dos tecnologías son las más ampliamente soportadas.

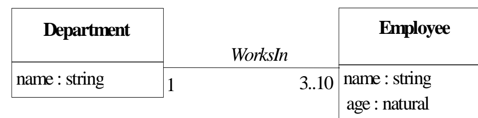
3. Evaluación de las herramientas

Las diferentes herramientas a evaluar se han clasificado en diferentes categorías: herramientas CASE, herramientas MDA, métodos completos de generación de código y herramientas OCL. Para cada grupo se han escogido las herramientas que, en nuestra opinión, son más representativas o que ofrecen un mejor soporte a la generación de RI.

Evidentemente, la clasificación de una herramienta en uno de estos tipos es un poco arbitraria ya que algunas herramientas podrían estar en varios de ellos a la vez.

Siguiendo los criterios de la anterior sección, se ha evaluado la generación de RI tanto para Java como para una base de datos relacional. Para cada tecnología se ha estudiado la expresividad permitida y, cuando ésta es representativa, la eficiencia de la implementación generada.

Como ejemplo a lo largo de la evaluación usaremos el esquema conceptual de la Figura 1. Este EC incluye (además de las restricciones de cardinalidad) tres restricciones textuales que comprueban que los empleados sean mayores de 16 años, que todos los departamentos tengan al menos 3 empleados mayores de 45 años y que no haya dos empleados con el mismo nombre.



context Employee **inv** ValidAge: self.age>16

context Department **inv** SeniorEmployees:
self.employee->select(e| e.age>45)->size()>=3

context Employee **inv** UniqueName:
Employee.allInstances()->isUnique(name)

Figura 1. Ejemplo de esquema conceptual

3.1. Herramientas CASE

Aunque al principio su principal función era facilitar la modelización de los sistemas software, hoy en día todas las herramientas CASE incluyen capacidades más o menos potentes de generación de código a partir de los modelos especificados por el diseñador.

Evidentemente es imposible evaluar todas las herramientas existentes (ver [9] para una lista exhaustiva) por lo que hemos seleccionado las siguientes: *Poseidon*, *Rational Rose*, *MagicDraw, Objecteering/UML* y *Together*.

A continuación comentamos en detalle las diferentes herramientas.

- *Poseidon* [10] es la extensión comercial de *ArgoUML* [11]. La generación de RI para la tecnología Java es muy pobre. No permite la definición de RI en OCL y tampoco tiene en cuenta las restricciones de cardinalidad. Sólo distingue entre multiplicidades 1 y superior a 1. De hecho, cuando la multiplicidad es superior a 1 no se controla ni que los objetos del atributo multivaluado generado sean del tipo correcto (ver la generación de la clase *Department* en la Figura 2). En algunas versiones *Poseidon* incorpora la herramienta *Dresden OCL* (ver sección 3.4) para ampliar su capacidad.

  La generación de las tablas relacionales también es bastante primitiva y no permite incorporar ninguna de las restricciones de

integridad del esquema, aparte de las claves primarias que hay que indicar marcando esta propiedad explícitamente en su definición.

```
public class Department {
  private string name;
  public java.util.Collection employee = new
java.util.TreeSet();
}
```

Figura 2. Clase *Departament* generada por *Poseidon*

- *Rational Rose* [12]. La generación para Java es similar a la de *Poseidon*. La generación para BDR es un poco más potente ya que permite definir ciertas propiedades adicionales en las clases y atributos del modelo. Por ejemplo permite definir la restricción *ValidAge* como propiedad del atributo *age* (ver Figura 3), cosa que provoca que la creación de la tabla *Employee* incluya la restricción *check(age>16)* para controlar la edad de los empleados.

  Recientemente ha aparecido un plug-in [13] para *Rational Rose* para permitir la especificación de todo tipo de restricciones OCL, aunque no se tienen en cuenta para la posterior generación de código.

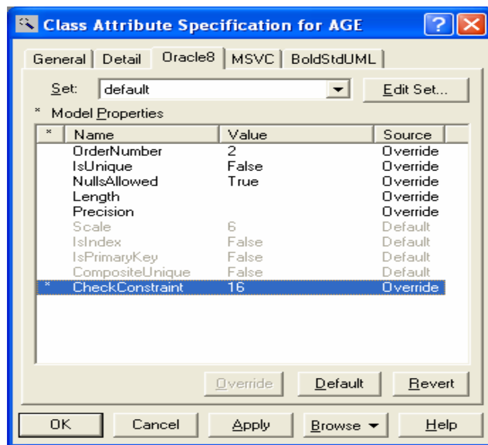


Figura 3. Propiedades de *Age* en Rational Rose

- *MagicDraw* [14] ofrece un soporte aún más pobre para la tecnología Java, ya que incluso genera incorrectamente las multiplicidades '*' de las asociaciones. No obstante, incorpora una característica interesante por lo que respecta a la generación de la BDR. Ofrece un tipo de diagrama específico para definir esquemas de bases de datos (de hecho no es más que un *profile* de UML predefinido). De esta forma se puede anotar el diagrama de clases con toda la información necesaria, incluyendo claves primarias, foráneas, restricciones de unicidad y comprobaciones sobre atributos.

  En la Figura 4 se muestra el esquema de base de datos correspondiente al EC de la Figura 1 una vez anotado con este profile, incluyendo las claves primarias de cada tabla, la clave foránea de empleado a departamento y la restricción *ValidAge*. Las otras restricciones no pueden especificarse ya que las BDRs no ofrecen ninguna construcción específica para comprobarlas (y la herramienta no es capaz de generar tampoco ningún tipo de código para hacerlo)

  El diagrama de la figura 4 se podría considerar como el PSM (Platform-Specific Model) del EC inicial (que seria a su vez el PIM (Platform-Independent Model).

  Además, la herramienta permite la generación semiautomática de este PSM a partir del PIM inicial.

  Finalmente, merece la pena destacar que aunque permite la especificación de restricciones OCL después las omite completamente en la generación de código posterior. Por ejemplo, en la transformación de PIM a PSM para el esquema de la Figura 1 no es capaz de transformar la restricción *ValidAge* en un *check* sobre el atributo *age* en el PSM, hay que indicar la restricción de nuevo en el PSM.

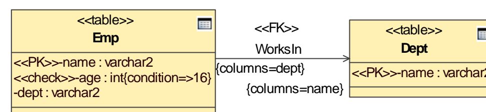


Figura 4. PSM para BDR de *MagicDraw*

- *Objecteering/UML* [15] ofrece como novedad respecto a las herramientas anteriores el hecho de permitir (y generar) cualquier tipo de multiplicidad en las asociaciones. Cuando genera código Java usa una librería de clases predefinida para realizar estas comprobaciones. Para la generación en una BDR genera un conjunto de triggers encargados de hacer las comprobaciones. Por ejemplo, el trigger de la Figura 5, mira antes de permitir la inserción

de un nuevo empleado que no existan ya 10 empleados para ese departamento. Si es el caso genera una excepción. El modelo de partida para la generación de la BDR es, como en *MagicDraw*, un PSM que puede obtenerse del EC inicial.

Como las anteriores tampoco permite la generación de restricciones especificadas en OCL.

```
CREATE TRIGGER TI_dept_emp_INSERT
 ON employee
 FOR INSERT
 AS
 IF NOT((SELECT COUNT(*)
     FROM employee, inserted
     WHERE employee.department =
             inserted.department) <= 10)
 BEGIN
   ROLLBACK TRANSACTION
   RAISERROR 20501
    "dept_emp : May not insert element, dept_emp_FK
maximum cardinality constraint violation"
 END
```

Figura 5. Trigger que controla la multiplicidad máxima

- Together [16] sigue una línea un poco diferente para la generación de bases de datos. Es capaz de generar las tablas desde un diagrama de clases pero recomienda la utilización de un diagrama ER para modelar el esquema de la base de datos más correctamente. Por lo demás, los resultados son parecidos a los de *Rational Rose*.

3.2. Herramientas MDA

Herramientas cuyo énfasis principal es dar soporte a la transformación de los modelos definidos por el diseñador de forma independiente de la tecnología (PIMs) a modelos dependientes de tecnología (PSMs) para pasar de éstos al código final. Han aparecido conjuntamente con la popularización de la propuesta MDA

En esta sección evaluamos dos de las herramientas MDA más conocidas: *ArcStyler* y *OptimalJ*.

ArcStyler [17] no está enfocada a la generación de bases de datos. La generación de restricciones de multiplicidad en Java es como la comentada anteriormente para *Poseidon* con la salvedad de que incluye la generación de operaciones que permiten incluir y eliminar objetos de los nuevos atributos (multivaluados) creados. Aunque permite especificar restricciones OCL (utiliza como entorno visual de modelización *MagicDraw*) la generación de las mismas con la herramienta *Dresden OCL* (también incorporada en *ArcStyler*) todavía no funciona correctamente.

OptimalJ [18] genera aplicaciones J2EE, donde toda la lógica del negocio (incluyendo la comprobación de RI) se concentra en las clases Java (Enterprise Java Beans, en este caso).

Da una gran importancia a las reglas de negocio pero sólo permite definir restricciones sobre el valor de los atributos y usando únicamente constantes o expresiones regulares. Para expresiones más complejas hay que escribir el código Java directamente.

3.3. Métodos para la generación de código

En esta sección incluimos dos de los principales representantes (*OO-Method* y *WebML*) de los métodos que aspiran a la generación completa del código de un sistema (tanto la parte estática como dinámica e incluso la interfaz gráfica) a partir de una especificación totalmente independiente de la tecnología final.

Dentro de esta categoría también recogemos la propuesta *Executable UML* aunque en este caso no se preocupa de la interfaz gráfica.

OO-Method [19] (y su extensión para la generación de aplicaciones Web [20]) se basa internamente en el lenguaje formal OASIS, aunque permite la especificación de esquemas conceptuales en UML.

Permite la definición de restricciones de integridad en un lenguaje similar al OCL. Estas restricciones se comprueban sobre las clases Java que representan el dominio de la aplicación. Después de ejecutar un método de una clase se comprueban todas sus restricciones asociadas (y no sólo las restricciones afectadas por ese método). Para comprobar las restricciones se ejecuta una operación que evalúa la condición de cada restricción tal y como ha sido definida por el usuario (ver Figura 6).

Las restricciones pueden incluir operadores de agregación pero no pueden definirse restricciones a nivel de clase. La implementación comercial del método se puede encontrar en [21].

```
Protected void checkIntegrityConstraints()
throws Error
{
  if (! (age>16))
  throw new error ("Constraint Violation.
Invalid age");
}
```

Figura 6. Comprobación de la restricción ValidAge

WebML [22] está especializado en la generación de aplicaciones web. Su soporte para restricciones de integridad es muy limitado. De hecho simplemente permite la definición de un *validity predicate*, que es una expresión booleana que permite comprobar la validez del valor entrado por el usuario en una de las páginas web que forman la interfaz de la aplicación (por ejemplo, comprobando que un cierto valor sea positivo o que esté dentro de un rango).

Por el contrario sí que permite la definición de elementos derivados, especificando su regla de derivación en OCL que después convierte en una vista SQL. No tendría que ser difícil aprovechar esta característica para controlar restricciones de integridad al estilo de lo que hace la herramienta *OCLtoSQL* (ver la sección 3.4).

Executable UML [23] propone especificar el esquema UML de forma que pueda ejecutarse directamente mediante su transformación interna a una tecnología concreta de implementación de forma transparente al usuario. Como característica principal adopta el uso de lenguajes de acciones para la definición de los métodos de las clases.

Permite la definición de ciertos tipos de restricciones predefinidos: cardinalidad, atributos identificadores (su valor tiene que ser único), restricciones sobre los valores de los atributos y restricciones sobre asociaciones cíclicas (por ejemplo que los objetos obtenidos navegando por una asociación sean los mismos o un subconjunto de los obtenidos a través de otra navegación).

Para su comprobación, éstas y cualquier otra restricción que queramos especificar tienen que definirse usando un *Action Language* (ver Figura 7). La principal herramienta que implementa estas ideas es *BridgePoint* [24] aunque con un soporte para restricciones de integridad todavía menor. Otras, como *iUML* [25] no incluyen ningún tipo de soporte. Ambas herramientas están muy enfocadas a la generación de aplicaciones de tiempo real y/o embebidas.

```
select many employees from instances of
Employee
Where selected.name==self.name
Return (cardinality employees)==0)
```

Figura 7. *UniqueName* definida con Action Language

3.4. Herramientas OCL

A continuación evaluamos herramientas dedicadas específicamente a la definición y generación de restricciones OCL.

Dresden OCL [26] es sin lugar a dudas la más conocida. Permite la generación de las clases Java correspondientes al diagrama de clases, incluyendo las restricciones de integridad definidas usando toda la expresividad del OCL, excepto el operador *allInstances*, que no genera correctamente.

Las restricciones se comprueban sólo después de cambios en el valor de los atributos (las asociaciones aparecen transformadas en atributos) incluidos en la definición de la restricción. Esto representa una mejora de eficiencia respecto a métodos anteriores que las comprobaban después de cualquier cambio pero, como se demuestra en [27], sigue siendo insuficiente ya que no cualquier tipo de modificación sobre los atributos puede violar la restricción. Por ejemplo, la restricción *SeniorEmployees* puede violarse cuando eliminamos un empleado del departamento pero no cuando lo añadimos.

OCLtoSQL es una herramienta incluida en el anterior kit y basada en el método propuesto en [28]. Permite la generación de una BDR a partir de un diagrama de clases con restricciones OCL. Para cada restricción crea una vista de forma que si al finalizar una transacción la vista no está vacía significa que la transacción ha violado la restricción representada por la vista. Por ejemplo, la Figura 8 muestra la vista correspondiente a la restricción *ValidAge*.

Las vistas se comprueban después de cada cambio sobre alguna de las tablas que se referencian en la vista (por lo tanto aunque se modifique sólo el nombre del empleado se controlará igualmente que no se viole la restricción *ValidAge*). Además no son incrementales ya que cada vez examinan toda la información de la base de datos y no sólo los datos modificados (en este caso se comprueban las

edades de todos los empleados y no sólo los modificados durante la transacción).

Las posibilidades de *Octopus* [29] son más limitadas. Para cada RI crea una nueva operación en la clase Java que representa el contexto utilizado en la definición de la RI. Para comprobar la RI sólo hay que ejecutar la operación. El cuándo ejecutar la operación se deja a criterio del programador, la herramienta no da ningún tipo de soporte al respecto. De la misma manera permite controlar las multiplicidades de las asociaciones.

```
create or replace view ValidAge as
(select * from EMPLOYEE SELF
 where not (SELF.AGE > 16));
```

Figura 8. Vista para la restricción ValidAge

La potencia de *OCLE* [30] y *KMF* [31] es similar a la de Octopus.

BoldSoft [32] es otra herramienta importante, aunque más pensada para la definición de elementos derivados que para la definición de RI. Permite ejecutar una expresión OCL tanto sobre los objetos en memoria principal como sobre los objetos guardados de forma persistente en una base de datos. En este último caso no permite toda la expresividad del OCL (por ejemplo no admite los operadores *count*, *collect*, *difference*, *asSet*, *asBag* …).

4. Características deseables de todo método de generación de RI

El objetivo de esta sección es presentar una serie de características, no proporcionadas actualmente por las herramientas y métodos existentes, pero que seria deseable que estuvieran presentes en todo método automático de generación de RI.

Aparte de las dos características básicas (expresividad y incrementalidad) también definimos las características de adecuación a la tecnología, independencia de tecnología y momento de la comprobación.

A continuación vemos en detalle cada una de ellas:

1. Expresividad: El método tendría que saber tratar toda la expresividad del lenguaje OCL, con lo que sería posible definir cualquier tipo de RI (estática).
2. Eficiencia: La generación del código para la comprobación de las RI debería verificar las RI sólo cuando sea estrictamente necesario y de una forma incremental.
3. Adecuación a la tecnología: Para conseguir una mayor eficiencia, el método tendría que saber aprovechar las características propias de la tecnología para la cual estamos generando la implementación.

   Por ejemplo, la generación para Java podría omitir las RI de *disjoint* (definen que la intersección entre los objetos de dos clases diferentes es vacía) ya que Java no admite clasificación múltiple y por lo tanto es imposible que dos clases no sean disjuntas.

   La misma idea se aplica a la bases de datos. Las BDRs ofrecen construcciones específicas como *primary key*, *check* (comprobaciones sobre atributos) o *unique*, por lo que si detectamos que alguna de las RI especificadas en OCL puede asimilarse a alguna de estas construcciones es preferible no tratarla nosotros mismos sino dejar que lo haga el Sistema Gestor de la Base de Datos (SGBD). Es razonable pensar que lo hará más eficientemente (por ejemplo preparando las estructuras internas adecuadas). El estándar teórico también define las *assertions* pero no están soportadas en los SGBDs actuales.
4. Independencia de la tecnología: Sería deseable que, como mínimo, las primeras etapas del método fueran independientes de la tecnología final. De esta forma se podría reaprovechar el método para generar las RI en diversas tecnologías. Sólo el último paso (la generación del código en sí) tiene que ser obligatoriamente dependiente (y tener en cuenta lo comentado en el punto 3).
5. Momento de la comprobación: En general, hay dos posibilidades en relación a cuando comprobar una RI. Se puede comprobar después de cada modificación de la BI (comprobación inmediata) o al finalizar la transacción u operación (comprobación diferida). Por ejemplo, el valor de un atributo puede cambiar varias veces a lo largo de una transacción por lo que una restricción sobre el rango de valores que pueda tomar es mejor comprobarla al final de la restricción. En cambio una restricción de unicidad generalmente es mejor comprobarla de forma inmediata (es difícil que el valor de un atributo que tiene que ser único cambie más de una vez a lo largo de la transacción).

El método tendría que permitir definir para cada restricción el momento de su comprobación.

5. Conclusiones

Con la aparición de MDA, la generación automática de código se ha vuelto a poner de plena actualidad.

En este artículo se ha estudiado qué posibilidades ofrecen las herramientas actuales respecto a la generación automática de restricciones de integridad. En estos momentos el panorama es bastante desolador. La mayoría de herramientas sólo permiten generar restricciones de integridad equivalentes a las construcciones ofrecidas por la tecnología final (como restricciones de unicidad y comprobaciones sobre atributos). Incluso en este caso, las restricciones tienen que indicarse explícitamente en el modelo, no se identifican automáticamente a partir de las expresiones OCL de las RI ya definidas previamente.

Algunas (pocas) herramientas sí son capaces de generar RI a partir de la definición en OCL. No obstante la comprobación generada es muy ineficiente. Como máximo intentan, sin conseguirlo completamente, comprobar las restricciones sólo después de cambios en la base de información que puedan afectar la restricción. Ninguna es capaz de realizar una comprobación incremental, se limitan a comprobar directamente la condición de la RI tal y como la define el diseñador.

Es evidente que aún queda mucho por hacer hasta conseguir una generación automática eficiente de las RI especificadas en un esquema conceptual. Hasta ese momento será discutible la afirmación de muchas de las herramientas acerca de que son capaces de generar automáticamente el 100% del código de la aplicación.

Como trabajo futuro también sería interesante aplicar este mismo estudio a las herramientas que permiten la definición de Domain-Specific Languages (como [33], [34] o [35] entre otras). Estas herramientas permiten la definición de nuevos metamodelos y comprueban que los modelos definidos por el diseñador sean consistentes con ellos. Esto implica verificar que los modelos no violan las RI definidas como parte del metamodelo (también llamadas *well-formedness rules*). En esquemas conceptuales grandes, la eficiencia de esta verificación adquiere una gran relevancia.

Agradecimientos

Este trabajo ha sido realizado con el apoyo del Ministerio de Ciencia y Tecnología, proyecto TIC2002-00744.

Referencias

[1] D. Teichroew, "Methodology for the Design of Information Processing Systems", presented at Fourth Australian Computer Conference, Adelaida, 1969.
[2] OMG, "MDA Guide Version 1.0.1", 2003.
[3] OMG, "UML 2.0 Superstructure Specification", OMG Adopted Specification (ptc/03-08-02), 2003.
[4] ISO/TC97/SC5/WG3, "Concepts and Terminology for the Conceptual Schema and Information Base", ISO 1982.
[5] OMG, "UML 2.0 OCL Specification", OMG Adopted Specification (ptc/03-10-14), 2003.
[6] D. W. Embley, D. K. Barry, and S. Woodfield, *Object-Oriented Systems Analysis. A Model-Driven Approach*: Yourdon, 1992.
[7] C. Türker and M. Gertz, "Semantic integrity support in SQL:1999 and commercial (object) relational database management systems", *The VLDB Journal*, vol. 10, pp. 241-269, 2001.
[8] A. Gupta and I. S. Mumick, "Maintenance of materialized views: problems, techniques, and applications", in *Materialized Views Techniques, Implementations, and Applications*: The MIT Press, 1999, pp. 145-157.
[9] Object by Design, "List of UML tools," 2005.
[10] Gentleware, *Poseidon for UML v. 3.1*, http://www.gentleware.com
[11] *ArgoUML v. 0.18.1*, http://argouml.tigris.org/
[12] Rational Software, *Rational Rose*, http://www-306.ibm.com/software/rational/
[13] EmPowerTec, *OCL-AddIn for Rational Rose*, http://www.empowertec.de/products/rational-rose-ocl.htm
[14] No Magic Inc., *MagicDraw UML v. 9.5*, http://www.magicdraw.com/
[15] Softeam, *Objecteering/UML v. 5.3*, http://www.objecteering.com/products.php

[16]Borland, *Borland® Together® Designer 2005*, http://www.borland.com/together/designer/index.html
[17]Interactive Objects, *ArcStyler v.5*, http://www.io-software.com/products/arcstyler_overview.jsp
[18]Compuware, *OptimalJ*, http://www.compuware.com/products/optimalj/
[19]O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming", *Information Systems*, vol. 26, pp. 507-534, 2001.
[20]J. Fons, V. Pelechano, M. Albert, and Ó. Pastor, "Development of Web Applications from Web Enhanced Conceptual Schemas", presented at 22nd Int. Conf. on Conceptual Modeling (ER'03), 2003.
[21]CARE technologies, *OLIVANOVA*, http://www.care-t.com/
[22]S. Ceri, P. Fraternali, and A. Bongio, "Web Modeling Language (WebML): a modeling language for designing Web sites", *Computer Networks*, vol. 33, pp. 137-157, 2000.
[23]S. J. Mellor and M. J. Balcer, *Executable UML*: Addison-Wesley, 2002.
[24]Accelerated Technology, *Nucleus BridgePoint Development Suite*, http://www.acceleratedtechnology.com/embedded/nuc_bridgepoint.html
[25]K. Carter, *iUML 2.2*, http://www.kc.com/products/iuml/index.html
[26]*Dresden OCL Toolkit*, http://dresden-ocl.sourceforge.net/index.html
[27]J. Cabot and E. Teniente, "Determining the Structural Events that May Violate an Integrity Constraint", presented at 7th Int. Conf. on the Unified Modeling Language (UML'04), 2004.
[28]B. Demuth, H. Hussmann, and S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications", presented at 4th Int. Conf. on the Unified Modeling Language (UML 2001), 2001.
[29]Klasse Objecten, *Octopus: OCL Tool for Precise Uml Specifications*, http://www.klasse.nl/english/research/octopus-intro.html
[30]Babes-Bolay University, *Object Constraint Language Environment 2.0.*, http://lci.cs.ubbcluj.ro/ocle/
[31]Kent Modelling Framework project, *Kent OCL Library*, http://www.cs.kent.ac.uk/projects/kmf/
[32]Borland, *Bold for Delphi*, http://info.borland.com/techpubs/delphi/boldfordelphi/
[33]MetaCase, *MetaEdit+*, http://www.metacase.com/mep/
[34]J. d. Lara and H. Vangheluwe, "AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling", presented at Fundamental Approaches to Software Engineering (FASE'02).
[35]M. Alanen and I. Porres, "Coral: A Metamodel Kernel for Transformation Engines", in *2nd European Workshop on Model Driven Architecture*, 2004, pp. 165-170.