Capítulo 3

EL LENGUAJE OCL

Anna Queralt
Ernest Teniente

3.1. INTRODUCCIÓN

Un lenguaje gráfico de definición de modelos, como podría ser UML, proporciona una idea general e intuitiva, generalmente fácil de comprender por parte del usuario, de los conceptos del dominio de referencia que se está modelando (ya sean conceptos del mundo real, componentes software, etc.) y de las relaciones entre ellos. Sin embargo, estos lenguajes no son lo suficientemente expresivos como para definir toda la información relevante de este dominio de referencia.

Por ejemplo, los conceptos relevantes de una liga de fútbol profesional se podrían modelar mediante el diagrama de clases de la Figura 1. A efectos de esta presentación, podemos suponer que este diagrama es un Modelo Independiente de la Plataforma en el que las clases representan conceptos del mundo real, con independencia de cómo éstos serán implementados después mediante un desarrollo dirigido por modelos. El diagrama captura todos los conceptos necesarios de este dominio: los equipos de fútbol y sus jugadores; las jornadas de la liga; los partidos entre un equipo local y uno visitante y la jornada en que se juegan; los jugadores convocados y los que

juegan en cada partido; los acontecimientos que suceden en los partidos, etc. También nos muestra las propiedades de cada uno de estos conceptos.

Un equipo de futbol está formado por varios jugadores. Un partido se define por un equipo que juega como local y otro que lo hace como visitante. Además un partido se juega en una determinada jornada. Los jugadores pueden ser convocados a los partidos en los que juega su equipo. Cuando se hace efectiva la convocatoria de un jugador a un partido se establece su posición en el campo y se especifica si el jugador empezará como titular o no. En un partido se producen varios acontecimientos. Un acontecimiento puede ser una falta, un gol, un cambio, una tarjeta amarilla o una tarjeta roja. De los acontecimientos necesitamos saber el minuto en el que se producen y el jugador que los protagoniza. Si un acontecimiento es un cambio, debe registrarse el jugador sustituto además del jugador sustituido (el protagonista del cambio).

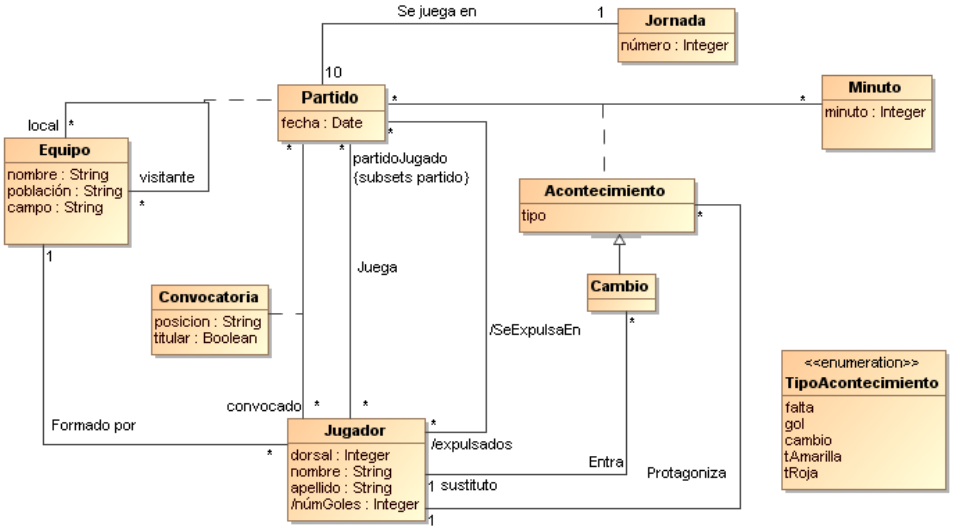

Figura 1: Diagrama de clases de una liga de fútbol

La Figura 1 proporciona una idea intuitiva, y a su vez formal, del dominio de referencia que se está especificando mediante este modelo. Sin embargo, no permite capturar aspectos tan relevantes como que los equipos local y visitante de un partido deben ser distintos, que los jugadores convocados como titulares en un partido tienen que jugar este partido o bien que los dos jugadores involucrados en un cambio (el sustituto y el que protagoniza el acontecimiento) deben pertenecer al mismo equipo. En general, estos aspectos hacen referencia a requisitos establecidos en el dominio de

referencia y también tienen que ser definidos en el modelo ya que de lo contrario la definición del dominio sería incompleta.

Por este motivo, el uso de lenguajes gráficos de modelado se complementa con el uso de otros lenguajes, normalmente textuales aunque basados en un formalismo lógico, que permitan definir esta información de manera precisa y no ambigua. El lenguaje OCL, acrónimo de *Object Constraint Language* en inglés o Lenguaje de Restricción de Objetos, es el más conocido de ellos. OCL ha sido propuesto por el Object Management Group (OMG 2012) y recientemente ha sido adoptado como estándar por el ISO/IEC (ISO 2012). Las características principales de OCL son las siguientes:

- *Es un lenguaje de definición de expresiones*, a partir de las cuales pueden definirse condiciones que un modelo debe satisfacer.
- *Es un lenguaje formal*, en el sentido que define expresiones con una semántica precisa y no ambigua, y que por lo tanto pueden ser evaluadas mediante herramientas automáticas.
- *Es un lenguaje tipado*, ya que cada expresión corresponde a un tipo de datos determinado y sólo se pueden combinar en OCL expresiones de tipos compatibles entre sí.
- *Es un lenguaje de especificación*, y no de programación.

El lenguaje OCL puede ser utilizado tanto para definir aspectos estructurales (o estáticos) de un modelo como para definir aspectos dinámicos (o de comportamiento). En la parte estructural de un modelo, OCL se usa para especificar las restricciones de integridad que el diagrama de clases debería satisfacer y también para definir la información derivada de este diagrama. En la parte del comportamiento, OCL se usa para definir el efecto de la ejecución de una operación sobre la base de información (mediante expresiones que especifican las precondiciones y las poscondiciones de dicha operación) y para definir consultas sobre la base de información.

OCL juega un papel importante en el Desarrollo de Software Dirigido por Modelos (MDE) ya que es utilizado en la creación de los Lenguajes Específicos de Dominio (DSL). Uno de los elementos importantes de un DSL es su sintaxis abstracta que se define mediante un metamodelo que especifica los elementos del lenguaje y las relaciones entre ellos, así como las reglas que establecen cuando un modelo del dominio está bien formado en relación a este metamodelo. OCL es el lenguaje más usado para escribir

estas reglas. El uso de OCL en el contexto del MDE se puede encontrar en (Warmer and Kleppe, 2003).

OCL (o algún lenguaje tipo-OCL) también es usado en el contexto del MDE para definir las transformaciones modelo a modelo o modelo a texto que se requieren para llevar a cabo el desarrollo de software mediante este enfoque dirigido por modelos. Ésta es otra de las razones de la importancia de OCL en MDD.

El objetivo de este capítulo es explicar, de manera intuitiva, como debería definirse una expresión OCL. Para ello nos basaremos en el diagrama de clases UML de la Figura 1 e iremos especificando cada uno de los aspectos adicionales de este diagrama que requieren del uso del lenguaje OCL. Concretamente, en la sección 3.2 describiremos el uso de OCL para definir restricciones de integridad. En la sección 3.3 trataremos la definición de la información derivada. En la sección 3.4 la definición de las operaciones de modificación y de consulta del esquema conceptual. En la sección 3.5 mencionaremos algunas de las herramientas de soporte al lenguaje OCL. Finalmente, en la sección 3.5 expondremos nuestras conclusiones.

El objetivo de este capítulo no es el de explicar la sintaxis exacta de todos los operadores que se pueden usar para definir una expresión OCL si no el de proporcionar una visión intuitiva del uso de OCL como complemento necesario de un lenguaje gráfico de (meta)modelado conceptual. Una especificación más formal de OCL se puede encontrar en (OMG 2012).

3.2. DEFINICIÓN DE RESTRICCIONES DE INTEGRIDAD

En OCL, una restricción de integridad se define como un invariante que todo estado de la base de información debe satisfacer. La expresión OCL que define este invariante se escribe en el contexto de una instancia de una clase determinada, la *instancia contextual*, y se define como una *propiedad de esta instancia*. Esta propiedad puede hacer referencia a los atributos de la instancia contextual o bien a una navegación iniciada a partir de la instancia contextual y realizada a través de las asociaciones del diagrama de clases. El tipo de la expresión que define una restricción debe ser Booleano, y debe especificar una condición que tiene que ser cierta para toda instancia contextual.

3.2.1 Identificación de la instancia contextual

La primera dificultad con la que nos enfrentamos al definir una restricción de integridad en OCL es la identificación de la instancia contextual. En algunos casos, esta tarea puede ser muy sencilla. Por ejemplo, si queremos definir un invariante para garantizar que la liga no puede incluir jugadores menores de quince años es evidente que la instancia contextual debe ser un objeto de la clase jugador. A partir de aquí, podemos generar ya la cabecera a partir de la cual podremos escribir la expresión que define el invariante deseado. En nuestro ejemplo, la cabecera sería:

```
context Jugador inv edadAdmisible:
```

edadAdmisible es el nombre que le damos a la restricción de integridad para poder referirnos a ella. La expresión que se defina a partir de este contexto debe ser cierta para todas las instancias de la clase *Jugador*. Habitualmente en esta expresión se utiliza la variable anónima *self* para referirse a cada una de estas instancias de manera individual. Entonces, podemos definir el invariante que nos ocupa de la manera siguiente:

```
context Jugador inv edadAdmisible:
            self.edad > 15
```

La subexpresión `self.edad` especifica un acceso al atributo *edad* de la instancia contextual *self*. Por lo tanto, el valor que se obtiene es de tipo entero. El valor obtenido se compara con la constante *15*, con lo cual el invariante evaluará a cierto si el jugador tiene más de quince años, y esta era la restricción que queríamos imponer en nuestro modelo.

En lugar de usar la variable *self*, las expresiones que definen los invariantes también pueden usar un nombre concreto si así se desea. Es suficiente con cambiar ligeramente la cabecera de la expresión:

```
context j:Jugador inv edadAdmisible:
            j.edad > 15
```

En general, puede haber más de una instancia contextual que nos permita definir una determinada restricción de integridad. Por ejemplo, para decir que los equipos locales y visitante de un partido tienen que ser diferentes podríamos usar tanto *Partido* como *Equipo* como instancia contextual. La selección de una clase u otra puede comportar que la expresión necesaria para definir la restricción de integridad sea más sencilla o compleja pero en

ambos casos se puede llegar a definir esta restricción, como veremos más adelante.

3.2.2 Navegación entre clases de objetos

En algunos casos, la expresión que define una restricción de integridad puede hacer referencia únicamente a algún atributo de la instancia contextual, como el ejemplo que acabamos de ver. De todos modos, en general esta expresión se define mediante una navegación iniciada a partir de la instancia contextual y realizada a través de las asociaciones del diagrama de clases.

Para navegar en OCL se usa la expresión: *objeto.nombreDeRol*, donde *objeto* hace referencia a la instancia contextual y *nombreDeRol* es una de las posibles destinaciones a la que se puede acceder mediante las asociaciones definidas en la clase de la instancia contextual. Si el nombre de rol no está definido y no hay ambigüedad posible, podemos usar el nombre de la clase destino empezando en minúscula para realizar esta navegación.

Por ejemplo, la restricción de integridad que impide que un mismo equipo juegue como local y como visitante en un partido de fútbol se podría definir de la manera siguiente:

```
context Partido inv equiposCorrectos:
            self.local <> self.visitante
```

`self.local` es una expresión que, partiendo de una instancia de la clase *Partido*, obtiene como resultado una instancia de la clase *Equipo* a través del rol *local*. Lo mismo sucede con la expresión `self.visitante` pero a través del rol *visitante*. Por lo tanto, el invariante se satisfará (y con él la restricción de integridad) si los dos equipos obtenidos mediante estas expresiones no son el mismo equipo.

Esta misma restricción se hubiera podido definir también considerando la clase *Equipo* para definir la instancia contextual:

```
context Equipo inv equiposCorrectos2:
            self.equipo[visitante] -> excludes(self)
```

Ahora la expresión `self.equipo[visitante]` define una navegación de una instancia de la clase *Equipo* a un conjunto de instancias de la misma clase, navegando a través del rol *visitante*. En este caso, al navegar por la asociación se obtienen un conjunto de instancias en lugar de una única instancia ya que la multiplicidad de la asociación para el rol especificado es

"*". Intuitivamente, el conjunto obtenido contiene todos los equipos que se han enfrentado a *self* como visitantes. `excludes` es una operación OCL que, aplicada sobre un conjunto de elementos (denotado por el símbolo `->` a la izquierda de la operación), evalúa a cierto si el objeto que se le pasa como parámetro no pertenece al conjunto de entrada. En este caso el invariante se cumplirá si *self* no se ha enfrentado a él mismo como visitante, que es la restricción que se pretendía definir.

En general, una navegación entre dos clases de objetos producirá un conjunto de objetos de la clase destino como resultado (excepto si la multiplicidad del rol por el que se está navegando es 1, con lo que se obtendría únicamente un objeto). Sin embargo, una concatenación de navegaciones produce como resultado una colección de objetos, denominada *bag* (o bolsa), que puede contener objetos repetidos.

Por ejemplo, la restricción de que el jugador que protagoniza un acontecimiento es uno de los convocados al partido se definiría como:

```
context Partido inv jugadorAdecuado:
 self.acontecimiento.jugador -> includesAll(self.convocado)
```

`self.acontecimiento.jugador` retorna una bolsa de objetos jugador ya que un mismo jugador puede haber protagonizado más de un acontecimiento en un mismo partido. `includesAll` es una operación que evalúa a cierto si la colección de todos los objetos que se le pasan como parámetro (en este caso los jugadores convocados al partido) pertenecen a la colección de entrada.

La operación `asSet()` permite convertir una bolsa de elementos en un conjunto mediante la eliminación de los objetos repetidos en la colección de entrada. El uso de esta operación no es necesario para definir la restricción anterior pero a veces se tiene que convertir una bolsa en un conjunto para que el resultado de la expresión sea correcto. Veremos un ejemplo cuando expliquemos el uso de OCL para definir la información derivada en la Sección 3.3.

3.2.3 Selección de un subconjunto de objetos

En algunas ocasiones, puede interesar no obtener todos los objetos resultantes de una navegación sino obtener únicamente un subconjunto de los mismos, que satisfacen una determinada condición. Esto es necesario,

por ejemplo, para definir la restricción de que todos los jugadores convocados como titulares para un partido deben jugar el partido:

```
context Partido inv titularesJuegan:
   self.convocatoria -> select(c|c.titular).convocado
                              -> includesAll(self.jugador)
```

`self.convocatoria` retorna todos los objetos *convocatoria* del partido *self*. De estos, nos interesan sólo los que son titulares con lo que tenemos que seleccionar un subconjunto de elementos del conjunto de entrada. Esto se consigue mediante la expresión `select(c|c.titular)`, dónde `c` es el nombre de variable usado para referirse a cada una de las convocatorias y `c.titular` es la condición booleana que `c` debe satisfacer para ser considerado. La última navegación de esta subexpresión, `.convocado`, permite obtener el conjunto de jugadores que han sido convocados como titulares al partido *self*. A partir de aquí, el invariante exige que los jugadores que juegan estén todos incluidos en este conjunto, mediante el uso de expresiones que ya hemos analizado con anterioridad.

3.2.4 Condición aplicable a todos los objetos de una colección

Una restricción OCL puede también hacer referencia a una propiedad compartida por todos los objetos obtenidos mediante una navegación a través de asociaciones del diagrama de clases. Por ejemplo, todos los jugadores convocados en un partido tienen que ser de uno de los dos equipos involucrados en el partido. La instancia contextual de esta restricción podría perfectamente ser el partido. A partir de aquí se pueden obtener los jugadores convocados y después exigir que cada uno de ellos pertenezca al equipo local o visitante del partido. Esta condición tiene que ser cierta para todos los jugadores del partido y esto se consigue en OCL mediante la operación de colecciones `forAll(expr)` que exige que la expresión `expr` sea cierta para todo elemento de la colección de entrada de la operación. Así pues, una forma de expresar esta restricción sería:

```
context Partido inv convocatoriaCorrecta:
   self.convocado -> forAll(j|
           j.equipo=self.local or j.equipo=self.visitante)
```

En OCL, del mismo modo que sucede en lógica matemática, exigir que una expresión sea cierta para todos los objetos de una colección es equivalente a que no exista ningún objeto de la colección tal que la expresión sea falsa. Así pues, la anterior restricción también se hubiera podido definir como:

```
context Partido inv convocatoriaCorrecta2:
   not self.convocado -> exists(j|
        j.equipo<>self.local and j.equipo<>self.visitante)
```

Una restricción puede depender también del número de objetos obtenidos como resultado de una navegación. En este sentido, OCL proporciona la operación `size()` para contar el número de elementos de una colección. Por ejemplo, en este dominio de aplicación es lógico exigir que un jugador no pueda tener más de dos tarjetas amarillas en un partido, ya que la segunda amarilla implica necesariamente la expulsión con lo que el jugador ya no puede continuar jugando. En OCL, esta restricción se podría definir como:

```
context Partido inv máximo2Amarillas:
self.convocado ->
  forAll(j|j.acontecimiento ->
          select(a|a.tipo=TipoAcontecimiento::tAmarilla)->
          size()<=2)
```

Este ejemplo sirve también para darnos cuenta de lo complicada que puede ser la expresión OCL usada para definir una restricción de integridad del modelo cuando se tienen que combinar varios tipos de operaciones.

También se requiere una expresión OCL compleja para definir la restricción que si a un jugador se le enseña una tarjeta roja o es cambiado en un partido no puede participar en ningún otro acontecimiento en un minuto posterior del mismo partido:

```
context Partido inv expulsadoNoProtagonizaMásEventos:
self.acontecimiento ->
  forAll(a1,a2| a1<>a2 and a1.jugador=a2.jugador and
         (a1.tipo=TipoAcontecimiento::tRoja or
          a1.tipo=TipoAcontecimiento::cambio) implies
          a1.minuto.minuto >= a2.minuto.minuto)
```

3.2.5 Navegación por jerarquías de especialización

Las expresiones que hemos considerado hasta ahora no han requerido ninguna navegación por jerarquías de especialización/generalización. Sin embargo, es bastante habitual que una restricción de integridad defina una propiedad que depende de estas jerarquías. OCL proporciona dos operaciones distintas, `o.oclIsTypeOf(t:Clasificador)` y `o.oclAsType(t:Clasificador)`. *oclIsTypeOf* devuelve cierto si el tipo de `o` y el de `t` son el mismo. *oclAsType* retorna como resultado el objeto `o` pero

visto como una instancia del tipo `t` si `t` es un subtipo del tipo actual de `o`. Si no lo es, el resultado de la expresión es indefinido.

Ejemplificaremos el uso de estos dos operadores mediante la especificación de la restricción que los dos jugadores que participan en un acontecimiento de cambio deben ser jugadores distintos que pertenecen al mismo equipo. En OCL, el invariante que expresa esta restricción podría definirse como:

```
context Acontecimiento inv cambioCorrecto:
self.oclIsTypeOf(Cambio) implies
   (self.jugador <> self.oclAsType(Cambio).sustituto and
    self.jugador.equipo =
           self.oclAsType(Cambio).sustituto.equipo)
```

La primera parte de la expresión `self.oclIsTypeOf(Cambio)` retorna un booleano que nos indica que esta expresión sólo se tiene que evaluar si el acontecimiento `self` es también un objeto de la subclase *Cambio*. Más adelante, necesitamos navegar a esta subclase mediante la operación `oclAsType(Cambio)` para poder acceder al rol `sustituto` que sólo está definido para los objetos de la misma.

La misma restricción se hubiera podido definir de forma más sencilla si se hubiera considerado la clase *Cambio* para definir la instancia contextual, en lugar de *Acontecimiento*:

```
context Cambio inv cambioCorrecto2:
   self.jugador <> self.sustituto and
   self.jugador.equipo = self.sustituto.equipo
```

Este ejemplo nos permite mostrar que en OCL se puede acceder directamente desde los objetos de una clase a todas las propiedades de sus superclases sin necesidad de utilizar la operación `oclAsType`. Así pues, siendo `self` un *Cambio*, podemos obtener el jugador que protagoniza el cambio mediante `self.jugador` sin necesidad de hacer `self.oclAsType(Acontecimiento).jugador`.

Otra constatación del ejemplo anterior es la importancia que tiene la identificación de la instancia contextual a la hora de requerir una expresión OCL más o menos simple para especificar una determinada restricción de integridad. Sin lugar a dudas, la expresión `cambioCorrecto2` es mucho más simple y fácil de entender que `cambioCorrecto` a pesar de que ambas expresiones definen la misma restricción.

3.2.6 Restricción de clave primaria de una clase de objetos

Para terminar la exposición de las operaciones más importantes que OCL proporciona para el tratamiento de las colecciones de objetos obtenidas como resultado de una navegación haremos referencia al uso de la operación `isUnique`. Esta operación se utiliza para definir que una determinada expresión o elemento aparece una única vez (o sea, es único) entre los elementos de la colección a la que se aplica. La operación `isUnique` puede utilizarse para definir de manera intuitiva la restricción de que un equipo no puede jugar más de un partido en una misma jornada:

```
context Jornada inv equiposNoRepetidos:
   self.partido.local -> union(self.partido.visitante)
                                          -> isUnique(nombre)
```

Intuitivamente, este invariante exige que el valor del atributo nombre sea único entre la colección de los equipos que han participado en la jornada `self` como locales o como visitantes. `union` es una operación OCL que permite realizar la unión entre dos colecciones de objetos que pueden haber sido obtenidos mediante una navegación, como sucede en este ejemplo.

3.2.7 Obtención de todos los objetos de una clase

En algunos casos, la comprobación de una restricción de integridad hace referencia a una propiedad de todos los objetos de una clase sin necesidad de efectuar ninguna navegación entre las asociaciones del diagrama de clases. El caso más habitual, con el que nos acostumbramos a encontrar en cualquier diagrama de clases, es la restricción que establece que un determinado atributo es la clave primaria de una clase o, de forma más general, la restricción de que dos objetos distintos de una misma clase no puedan tener idéntico valor para una determinada propiedad (atributo o nombre de rol accesible) o conjunto de propiedades.

OCL dispone de la operación `allInstances()` que aplicada a un nombre de clase permite obtener todos los objetos de esa clase. Con esta operación, la restricción de que el atributo *nombre* es la clave primaria de *Equipo* se podría definir como:

```
context Equipo inv nombreIdentifica:
    Equipo.allInstances() -> isUnique(nombre)
```

De manera similar se podrían definir las claves primarias de *Jornada* (número) y *Minuto* (minuto). Sin embargo, para definir que la clave

primaria de *Jugador* está formada por la concatenación de los atributos nombre y apellido deberíamos utilizar la expresión:

```
context Jugador inv nombreYApellidoIdentifica:
    Jugador.allInstances() -> forAll (j1,j2| j1<>j2 implies
          j1.nombre<>j2.nombre or j1.apellido<>j2.apellido)
```

Es importante tener en cuenta que no se debe abusar del uso de la operación `allInstances()`, tanto por motivos de comprensibilidad como para no facilitar la eficiencia del código o de los artefactos que puedan ser generados automáticamente a partir de esta expresión. De hecho, en muchas ocasiones se puede especificar una expresión equivalente (y preferible) sin necesidad de usar la operación `allInstances()`. Por ejemplo, si queremos especificar la restricción de que las instancias de la clase minuto van del 1 al 90 (suponiendo que esta es la duración máxima de un partido de fútbol), la manera más adecuada de definir esta restricción sería:

```
context Minuto inv minutoCorrecto:
    self.minuto>=1 and self.minuto<=90
```

En lugar de:

```
context Minuto inv minutoCorrecto:
    Minuto.allinstances()
                ->forall(m|m.minuto>=1 and m.minuto<=90)
```

En el primer caso, la expresión sería evaluada una única vez para cada una de las instancias de la clase *Minuto* mientras que en el segundo sería evaluada para todas las instancias de la clase cada vez que una de ellas fuera considerada como instancia contextual.

3.2.8 Resumen de operaciones aplicables a una colección

Para terminar esta sección, presentamos en la siguiente figura una tabla resumen de las operaciones OCL más habituales que se pueden aplicar sobre una colección de elementos, ya sea un conjunto o una bolsa (lista no ordenada que admite elementos repetidos), y el significado de cada una de ellas. Una formalización más precisa de dichas operaciones y alguna operación adicional de tipo colección se pueden encontrar en (OMG 2012).

Operación	Significado
`->size()`	Devuelve el número de elementos de la colección
`->includes(e)`	Devuelve cierto si e pertenece a la colección

`->includesAll(e)`	Devuelve cierto si tota instancia de e pertenece a la colección
`->excludes(e)`	Devuelve cierto si e no pertenece a la colección
`->excludesAll(e)`	Devuelve cierto si ninguna instancia de e pertenece a la colección
`->isEmpty()`	Devuelve cierto si la colección es vacía
`->notEmpty()`	Devuelve cierto si la colección no es vacía
`->exists(expr)`	Devuelve cierto si existe una instancia de la colección que cumple la expresión *expr*
`->forAll(expr)`	Devuelve cierto si tota instancia de la colección cumple la expresión *expr*
`->isUnique(expr)`	Devuelve cierto si no existen dos instancias distintas de la colección que tengan el mismo valor per la expresión *expr*
`->select(expr)`	Devuelve los elementos de la colección que cumplen la expresión *expr*
`->reject(expr)`	Devuelve los elementos de la colección que no cumplen la expresión *expr*
`->asSet()`	Devuelve la colección como conjunto (o sea, elimina repetidos si la colección era una bolsa)

Las operaciones de tipo colección se aplican siempre a una expresión que devuelve un conjunto de objetos obtenido como resultado de una navegación a través del diagrama de clases. Esta expresión debe especificarse a la izquierda de la -> y no ha sido mostrada explícitamente en la tabla anterior. De todos modos, en esta sección hemos mostrado numerosos ejemplos completos con usos correctos de algunas de estas operaciones.

3.3. DEFINICIÓN DE INFORMACIÓN DERIVADA

Un modelo UML admite la definición de dos tipos de información derivada: atributos y asociaciones. Un atributo o una asociación es derivado cuando su valor o sus instancias no tienen que estar explícitamente almacenados en el modelo ya que pueden ser derivados (o sea, inferidos o calculados) en

cualquier momento a partir del contenido de otros atributos o asociaciones (Olivé 2007).

Por ejemplo, el atributo número de goles de un jugador es derivado ya que su valor se puede calcular a partir de los eventos de tipo gol que ha protagonizado este jugador. De manera similar, la asociación que relaciona un partido con los jugadores que han sido expulsados en el mismo también es derivada ya que sus instancias también pueden ser calculadas a partir de los acontecimientos que se han sucedido en este partido y los jugadores involucrados en los mismos.

OCL ofrece también la posibilidad de definir las reglas de derivación necesarias para calcular el valor y el contenido de los atributos y las asociaciones derivados. De manera similar a las restricciones de integridad, el cálculo de una información derivada se define mediante una expresión OCL que se escribe en el contexto de una instancia determinada y cuya evaluación permitirá conocer el valor exacto de esta información derivada. A la hora de definir una regla de derivación en OCL es muy importante tener en cuenta que deben coincidir el tipo del atributo/asociación derivado con el tipo de la expresión usada para calcularlo.

La identificación de la instancia contextual de una información derivada es muy sencilla ya que ésta debe ser necesariamente la clase sobre la cual está definido el atributo o la asociación derivados. Además, la navegación a través de las asociaciones del diagrama de clases se efectúa de la misma manera que para las restricciones de integridad.

Así pues, el atributo derivado *numGoles* de un *Jugador* se puede definir mediante la siguiente expresión OCL:

```
context Jugador::numGoles:Integer
derive: self.acontecimiento ->
        select(a|a.tipo=TipoAcontecimiento::gol)->size()
```

numGoles es un atributo de tipo *entero* definido en la clase *Jugador*, tal y como se muestra en la cabecera (***context***) de esta expresión. La expresión definida a continuación de la palabra reservada ***derive*** es una expresión de tipo entero que al ser evaluada devolverá como resultado el número de goles marcados por el jugador *self* (instancia contextual, sobre la cual se define el atributo derivado cuyo valor es calculado mediante esta regla de derivación). El valor del atributo *numGoles* se calculará de la misma forma para cualquier objeto *self* de *Jugador*.

De manera similar, podríamos definir la asociación derivada SeExpulsaA entre un partido y sus jugadores:

```
context Partido::expulsados:Set(Jugador)
derive: self.jugador ->select(j|j.acontecimiento->
       includes(a|a.tipo=TipoAcontecimiento::tRoja))
```

La definición de una asociación derivada se efectúa mediante la definición de uno de los roles (derivados) de dicha asociación (*expulsados* en nuestro ejemplo). El tipo de la expresión definida para calcular esta asociación es un conjunto de objetos de la clase obtenida mediante la navegación a través de este rol.

3.4. DEFINICIÓN DE OPERACIONES

En los apartados precedentes hemos analizado el uso de OCL como lenguaje de soporte en la definición del esquema estructural de un modelo, consistente normalmente en un diagrama de clases. En UML, y en general en el desarrollo dirigido por modelos, no es suficiente con definir este esquema estructural sino que debe definirse también el esquema del comportamiento del modelo para especificar las consultas y las actualizaciones admitidas sobre la base de información definida por el esquema estructural. Esto se consigue mediante la definición de operaciones de consulta y de actualización del modelo, y OCL también puede ser utilizado para definir estas operaciones de manera precisa y no ambigua.

3.4.1 Definición de operaciones de consulta

Podemos utilizar una expresión OCL para indicar el resultado de una operación de consulta. Una operación de consulta no puede modificar el contenido de la base de información y, como es habitual en OCL, el resultado de la expresión OCL usada para definir el resultado de la consulta tiene que ser conforme al tipo de resultado esperado por la operación.

Por ejemplo, podríamos estar interesados en la definición de una operación de la clase *Equipo* para consultar el número de goles que un determinado jugador del equipo (identificado por su dorsal dentro del equipo) ha marcado en los partidos que el equipo ha disputado como local. Esta consulta se podría especificar en OCL de la manera siguiente:

```
context Equipo::obtenerGoles(dorsal:Integer):Integer
```

```
body: self.partido[local].acontecimiento
   -> select (a|a.tipo=TipoAcontecimiento::gol
                and a.jugador.equipo=self
                and a.jugador.dorsal=dorsal) -> size()
```

`obtenerGoles` es una operación de la clase *Equipo* que recibe como parámetro la información el *dorsal* del jugador sobre el que se está haciendo la consulta. La expresión OCL que define la navegación necesaria para obtener el resultado de la consulta se especifica en el contexto de la palabra reservada ***body***. En este ejemplo, se obtienen primero todos los acontecimientos de los partidos que el equipo `self` (instancia contextual de esta operación, que corresponde al equipo sobre el que se invoca la operación) ha disputado como local, se seleccionan a continuación los acontecimientos de tipo gol que ha protagonizado el jugador y se obtiene finalmente el número de objetos del conjunto resultante (mediante la operación `size`). Así pues, el resultado de esta expresión OCL es un entero, tal y como es requerido por la cabecera de la operación, que corresponde con la información que se deseaba obtener.

El resultado de la consulta anterior hace referencia a un valor de un único tipo, en este caso el de enteros. Sin embargo, es bastante frecuente que el resultado de una consulta haga referencia a distintos valores de información, que pueden ser además de tipos distintos. Este tipo de información compuesta se puede representar en OCL mediante tuplas (*tuples)*. Una *tupla* contiene varios elementos de información, cada uno de ellos referenciado por un nombre, y que pueden ser de tipos distintos. Una tupla es conforme a un tipo de tupla, *TupleType* en OCL, que debe ser declarado en la cabecera de la consulta para poder ser usado posteriormente en la expresión OCL que permite determinar el resultado de la consulta.

Por ejemplo, supongamos una operación de la clase *Jornada* que nos permita obtener información de los partidos de esa jornada con alguna expulsión. Para cada uno de estos partidos, la salida debería mostrar el nombre del equipo local, el nombre del equipo visitante y, para cada expulsión dentro del partido, el minuto en que se ha producido y el dorsal y el equipo del jugador expulsado. La cabecera de esta operación de consulta en OCL es la siguiente:

```
context Jornada::resumenExpulsiones():
 Set(TupleType(eLoc:String, eVis:String, infoJugadores:
    Set(TupleType(min:Integer, dors:Integer, eq:String))
```

Estamos definiendo una operación `resumenExpulsiones` de la clase *Jornada*, sin ningún parámetro de entrada y con una salida formada por

información compuesta. El contenido de esta información está definido por un tipo de tupla compuesto por tres partes, equipo local *eLoc*, visitante *eVis* e información de los jugadores expulsados *infoJugadores*, que se muestra repetidas veces (de ahí el uso de la palabra `Set` para definir que se trata de un conjunto de información). *infoJugadores* a su vez es otra información compuesta. En este caso, por el minuto en que se produce la expulsión *min*, el dorsal del jugador que la protagoniza *dors* y el nombre del equipo al que pertenece *eq*.

La definición de la expresión OCL tiene que ser capaz de manipular la estructura del *TupleType* para ir construyendo las tuplas que se requieren para mostrar al usuario la información de salida deseada. Una posible expresión que nos calculará esta resultado sería la siguiente:

```
body:
let partidos:Set(Partido) =
 self.partido -> select(p|p.acontecimiento
         -> select(a|a.tipo=TipoAcontecimiento::tRoja)
             -> notEmpty())
in
 partidos -> collect(p|Tuple
    {eLoc = p.equipo[local].nombre,
     eVis = p.equipo[visitante].nombre,
     infoJugadores = p.acontecimiento
         -> select(a|a.tipo=TipoAcontecimiento::tRoja)
         -> collect(a|Tuple
                     {min = a.minuto.minuto,
                      dors = a.jugador.dorsal,
                      eq = a.jugador.equipo.nombre})})
```

La expresión *let* permite definir una variable *partidos* que usamos para almacenar todos los partidos de la jornada sobre la que se invoca la operación en la que se ha producido alguna expulsión. La operación `notEmpty`, usada en la expresión OCL que identifica estos partidos, se aplica sobre una colección de elementos y devuelve cierto si la colección contiene algún elemento (o sea, si no es vacía). La variable *partidos* se utiliza después para ir coleccionando las tuplas requeridas por la salida. La obtención del contenido de cada uno de estas tuplas se realiza mediante navegaciones por el contenido del esquema estructural similares a las que hemos visto anteriormente en este capítulo.

3.4.2 Definición de operaciones de actualización

Una operación de actualización se define mediante un contrato que especifica el efecto que la ejecución de la operación produce sobre el contenido de la base de información. Este efecto se define mediante un conjunto de *precondiciones*, cada una de las cuales especifica una condición necesaria para que la operación se pueda ejecutar, y un conjunto de *postcondiciones*, que definen los cambios concretos que provoca la ejecución de la operación. OCL también se puede utilizar para definir las precondiciones y las postcondiciones de una operación (Larman 2002).

Por ejemplo, supongamos una operación de la clase *Partido* que permita registrar en la base de información los cambios que se producen durante un partido. Esta operación debería tener la información del cambio producido como parámetro (o sea, el minuto en que se produce, el equipo que realiza el cambio, el dorsal del jugador que es cambiado y el del jugador sustituto). Las únicas (pre)condiciones que debe controlar esta operación son que los jugadores involucrados en el cambio tienen que existir y pertenecer al equipo que realiza la sustitución (las otras condiciones posibles ya están garantizadas por el esquema estructural y por lo tanto no deben ser consideradas de nuevo en aras de la no redundancia (Queralt and Teniente 2006). Una vez garantizado que esto es así, la postcondición tiene que especificar los cambios realizados por la ejecución de la operación. En OCL, esta operación se podría especificar de la manera siguiente:

```
context Partido::nuevoCambio(min:Minuto, nomEq:String,
                             dors1:Integer, dors2:Integer)
pre: jug1Existe: Jugador.allInstances() ->
     exists (j|j.dorsal=dors1 and j.equipo.nombre=nomEq)
pre: jug2Existe: Jugador.allInstances() ->
     exists (j|j.dorsal=dors2 and j.equipo.nombre=nomEq)
post: Cambio.allInstances -> exists(c|
                   c.oclIsNew() and c.tipo=”cambio”
                   and c.partido=self
                   and c.minuto.minuto=min
                   and c.sustituto.dorsal=dors2
                   and c.sustituto.equipo.nombre=nomEq
                   and c.jugador.dorsal=dors1
                   and c.jugador.equipo.nombre=nomEq)
```

`jug1Existe` y `jug2Existe` son los nombres que le damos a cada una de las precondiciones de la operación para después podernos referir a cada una de ellas de forma individual si es necesario. La operación `oclIsNew` devuelve

cierto si el objeto sobre el que se aplica no existía en el estado de la base de información anterior a la ejecución de la operación y nos permite especificar que un nuevo objeto de la clase *Cambio* será creado como consecuencia de la ejecución de la operación. La asignación de los valores a los atributos de este objeto y sus asociaciones con los objetos relacionados se realiza de manera muy similar a las expresiones OCL que permiten obtener el resultado de una consulta.

En la operación anterior, también se hubieran podido pasar directamente el objeto `e:Equipo` como parámetro en lugar del string que representa su clave primaria y usar el hecho de que un equipo conoce a sus jugadores con lo que nos ahorraríamos el uso de la operación `allInstances()`. En este caso, la precondición `jug1Existe` se definiría de la manera siguiente:

```
pre: jug1Existe: e.jugador.dorsal -> includes(dors1)
```

Para finalizar este apartado, vamos a especificar la operación que permite a un jugador cambiar de equipo. Esta operación sólo se podrá llevar a cabo si el nuevo equipo existe en la base de información y si el jugador en cuestión no ha participado en ninguna convocatoria durante el campeonato. En este caso, no debe crearse ningún nuevo objeto como consecuencia de la ejecución de la operación ya que la única modificación que debe producirse en la base de información es la asignación del nuevo equipo al jugador y la eliminación del equipo actual. El contrato OCL de esta operación es el siguiente:

```
context Jugador::cambioEquipo(nombreEquipo:String)
pre: nuevoquipoExiste: Equipo.allInstances() ->
                         exists (e|e.nombre=nombreEquipo)
pre: noConvocado: self.convocatoria -> isEmpty()
post: self.equipo.nombre = nombreEquipo
```

En este caso, si se pasara un equipo como argumento en lugar de su clave primaria el contrato quedaría de la manera siguiente:

```
context Jugador::cambioEquipo(nuevoEq:Equipo)
pre: noConvocado: self.convocatoria -> isEmpty()
post: self.equipo = nuevoEq
```

En este nuevo contrato no hace falta comprobar que el nuevo equipo existe dado que este objeto ha sido ya pasado como parámetro.

3.4. HERRAMIENTAS DE SOPORTE

En la actualidad existen varias herramientas de soporte al lenguaje OCL. El objetivo de esta sección es el de presentar una lista, no exhaustiva, de herramientas que puedan ser de utilidad en el proceso de aprendizaje de UML y para comprender mejor la utilidad de OCL tanto en el contexto del MDE como del modelado conceptual.

La herramienta más conocida para escribir código OCL asociado a un metamodelo es OCLinEcore (OCLinEcore 2012) que forma parte del proyecto MDT de Eclipse/EMF. En el marco del proyecto MDT de Eclipse/EMF también encontramos la herramienta MDT/OCL (MDT/OCL 2012) que ha sido construida con el objetivo de proporcionar una implementación de metamodelos estándar industriales y para facilitar la construcción de herramientas para desarrollar modelos basados en estos metamodelos. En el ámbito de este mismo proyecto también se ha desarrollado la herramienta OCLTools (OCLTools) que proporciona facilidades de edición, generación de código, ejecución o depuración interactiva de expresiones escritas en OCL. Uno de los analizadores sintácticos más conocidos de OCL es la herramienta Dresden/OCL (Dresden 2012).

Además de las herramientas anteriores, también se han desarrollado herramientas para la especificación y la validación de esquemas conceptuales especificados en UML/OCL. Una de las más conocidas es USE (Gogolla, Büttner, Richters 2007) que permite animar el esquema conceptual para validar la especificación de los requisitos funcionales de un sistema de información. La herramienta permite crear y manipular instanciaciones concretas del esquema conceptual y comprobar automáticamente que dicha instanciación satisface las restricciones OCL del esquema.

También se han desarrollado varios prototipos para la verificación y/o la validación automática de un esquema conceptual, con la finalidad de comprobar automáticamente que el esquema es correcto y que representa adecuadamente los requisitos de los usuarios del sistema. Los más conocidos en la actualidad son HOL-OCL (Brücker and Wolff 2006), UML2Alloy (Anastasakis et al. 2007), UMLtoCSP (Cabot et al. 2007) o AuRUS (Queralt et al. 2010).

Finalmente, comentar que muchas de las herramientas CASE comerciales actuales proporcionan soporte para la definición, y en cierta medida el

análisis sintáctico, de restricciones de integridad OCL asociadas a un esquema conceptual UML. Dado el gran número de herramientas CASE que admiten esta funcionalidad y la elevada volatilidad de estas herramientas hemos preferido no mencionarlas explícitamente en este capítulo.

3.5. CONCLUSIONES

Los lenguajes gráficos de (meta)modelado, como por ejemplo UML, deben complementarse necesariamente con el uso de un lenguaje textual para expresar todos aquellos conceptos que no pueden expresarse de forma gráfica. El lenguaje OCL, recientemente adoptado como estándar por el ISO/IEC, es el más conocido en la actualidad. El objetivo de este capítulo ha sido el de proporcionar una visión intuitiva del uso de OCL, mediante su aplicación a un caso práctico, en aras de justificar la importancia de este lenguaje en el contexto del MDE y del desarrollo de software en general.

3.6. REFERENCIAS

Anastasakis, K.; Bordbar, B.; Georg, G.; Ray, I. (2007). Uml2Alloy: A Challenging Model Transformation. MoDELS 2007, pp. 436–450. Springer.

Brucker, A.D.; Wolff, B.(2006). The HOL-OCL book. Tech. Report 525, ETH Zurich.

Cabot, J.; Clarisó, R.; Riera, D. (2007). UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In ASE 2007, pp. 547–548. ACM,

ISO (2012): ISO/IEC 19505-2:2012 - OMG UML superstructure 2.4.1. http://www.iso.org/.

Dresden/OCL (2012). http://www.dresden-ocl.org/index.php/DresdenOCL.

Gogolla, M.; Büttner, F.; Richters, M. (2007). USE: a UML-based Specification Environment for Validating UML and OCL. Science of Computer Programming. 69 (1-3), pp. 27-34.

Larman, C. (2002). Applying UML and Patterns (2nd Edition). Prentice Hall.

MDT/OCL (2012). www.eclipse.org/modeling/mdt/?project=ocl.

OCLinEcore (2012). http://wiki.eclipse.org/MDT/OCLinEcore.

OCLTools (2012). http://wiki.eclipse.org/MDT-OCLTools.

Olivé, A. (2007). Conceptual Modeling of Information Systems. Springer.

OMG (2012): Object Constraint Language (OCL). http://www.omg.org/spec/OCL/

Queralt, A., Teniente, E. (2006) . Specifying the semantics of operation contracts in conceptual modeling. Journal on Data Semantics VII, volume 4244. Springer, pp. 33-56.

Queralt, A.; Rull, G.; Teniente, E.; Farré, C.; Urpí, T. (2010). AuRUS: Automated Reasoning on UML/OCL Schemas. Int. Conf. on Conceptual Modeling (ER 2010), pp. 438–444.

Warmer, A.; Kleppe, J. (2003). The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition). Addison-Wesley.

ÍNDICE ALFABÉTICO

Se deben marcar en el texto de todo el libro las palabras clave para referenciarlas en este índice siguiendo las indicaciones del manual de procedimiento editorial.

Autores

ANNA QUERALT

Ingeniera Informática y Doctora en Informática por la Universitat Politècnica de Catalunya. Ha impartido docencia en ingeniería del software y UML en la Facultat d'Informàtica de Barcelona y en la Universitat Oberta de Catalunya. Su investigación se ha centrado principalmente en el modelado conceptual en UML y OCL, así como en la validación automática de estos modelos. Actualmente es investigadora en el Barcelona Supercomputing Center-Centro Nacional de Supercomputación, donde se centra en el almacenamiento y gestión de datos.

ERNEST TENIENTE

Doctor en Informática por la Universitat Politècnica de Catalunya (UPC). Su docencia ha estado ligada durante muchos años al desarrollo de software y en particular a la especificación y al diseño de software con el uso de los lenguajes UML y OCL. Su investigación se ha centrado principalmente el modelado conceptual y en el razonamiento automático de esquemas UML/OCL y en el tratamiento de problemas relacionados con la actualización consistente de bases de datos. Es autor de numerosas publicaciones internacionales en congresos y revistas de reconocido prestigio y también forma parte con regularidad de comités de programa de conferencias en ingeniería del software y bases de datos. También dirige el grupo de investigación MPI de la UPC cuya actividad se centra actualmente en el modelado y el procesamiento de información.