CAPÍTULO 12. NOTACIONES FORMALES. LENGUAJE OCL.

Con la colaboración de Luis Reynoso, de la Universidad Nacional del Comahue (Argentina)

1. Introducción

El lenguaje OCL (*Object Constraint Language* o *Lenguaje de Restricciones sobre Objetos*) se utiliza como complemento a UML. OCL permite describir de manera más precisa y sin ambigüedades los sistemas descritos con UML.

OCL se utiliza para describir los siguientes tres tipos de restricciones (ya se mencionaron en la página 75 al hablar de los Contratos):

- Precondiciones: para un operación de una cierta clase, una *precondición* especifica una condición que debe ser cierta antes de ejecutar la operación.
- Postcondiciones: para una operación de una cierta clase, una *postcondición* representa una condición que debe ser cierta después de ejecutar la operación.
- Invariantes: una *invariante* para una clase, tipo o interfaz representa una condición que siempre debe ser satisfecha por todas las instancias de tal clase, tipo o interfaz. Por tanto, siempre deben ser ciertas.

Las restricciones se escriben en OCL utilizando una notación específica, en la que intervienen tipos (en el sentido de *clases*), cada uno con sus operaciones y atributos. En la siguiente sección se presentan los tipos predefinidos en OCL; después, se ilustran la notación añadiendo algunas diferentes tipos de restricciones a un ejemplo.

2. Tipos predefinidos de OCL

2.01 Tipos básicos no escalares

En OCL existe un conjunto de tipos básicos, que son independientes del modelo de objetos que se esté desarrollando, como *Real, Integer, Boolean* y *String.* Además, todos los tipos tanto de UML como de OCL tienen un tipo, que es una instancia de un tipo especial de OCL llamado *OclType.* Todo *OclType* tiene los atributos y operaciones que se muestran en la Tabla 7:

Atributos de *OclType*		
Atributo u operación	**Tipo**	**Descripción**
name	String	nombre del tipo (en la clase *Persona*, este atributo vale "Persona")
attributes	Set(String)	conjunto formado por los nombres de los atributos del tipo (por ejemplo, si el tipo es la clase *Persona*, en sus *attributes* tendríamos *mNombre, mApellidos, mNIF*, etc.)
associationEnds	Set(String)	conjunto formado por los nombres de los tipos a los que se puede navegar mediante asociaciones y agregaciones desde el tipo en el que nos encontramos (en la Figura 44, página 50, desde *Tarjeta* podemos navegar hasta *Cuenta*)
operations	Set(String)	conjunto formado por todas las operaciones de este tipo
supertypes	Set(OclType)	conjunto formado por los supertipos directos ("padres" de este tipo)
allSupertypes	Set(OclType)	conjunto formado por todos los supertipos de este tipo (padres, abuelos, etc.)
allInstances	Set(type)	conjunto de todas las instancias de *type* y de sus supertipos

Tabla 7. Atributos de *OclType*

Además, todo tipo de OCL es un subtipo de *OclAny*, que es el supertipo de todos los tipos de un modelo de objetos. Por tanto, todos los miembros definidos en *OclAny* los tienen también todos los tipos que vayamos definiendo:

Atributos y operaciones de *OclAny*		
Atributo u operación	**Tipo**	**Descripción**
objeto = (x : OclAny)	Boolean	Operador de comparación, que devuelve *true* si los dos objetos que comparamos son iguales y *false* en caso contrario. A todos los tipos de nuestros modelos podemos aplicarle el operador =
objeto <> (x : OclAny)	Boolean	Operador de comparación, que devuelve *true* si los dos objetos que comparamos son distintos y *false* si son iguales. A todos los tipos de nuestros modelos podemos aplicarle el operador <>
objeto.oclType	OclType	Tipo de este objeto
objeto.oclIsKindOf(type:OclType)	Boolean	Devuelve *true* si *type* es supertipo del tipo de *objeto* o el mismo tipo
objeto.oclIsTypeOf(type:OclType)	Boolean	Devuelve true si el tipo del objeto es *type*

Tabla 8. Atributos y operaciones de *OclAny*

Por último, OCL define también el tipo *OclExpression*, que representa el tipo de cualquier expresión de OCL (o sea: puesto que toda expresión tiene un tipo, toda expresión es un objeto de clase/tipo *OclExpression*). El único miembro del tipo *OclExpression* se muestra en la siguiente tabla:

El único miembro de *OclExpression*		
Atributo u operación	**Tipo**	**Descripción**
expr.evaluationType	OclType	Tipo del objeto obtenido al evaluar *expr*

Tabla 9. El único miembro de *OclExpression*

2.02 Tipos básicos escalares

Respecto de los tipos básicos "escalares", éstos y sus operaciones se muestran en la siguiente tabla:

Tipo	Operaciones
Integer	+(Integer):Integer; -(Integer):Integer; *(Integer):Integer; /(Integer):Integer abs():Integer; div(Integer):Integer mod(Integer):Integer; max(Integer) min(Integer):Integer; Además: < > <= >= = <>
Real	+(Real):Real -(Real):Real *(Real):Real /(Real):Real abs():Real floor():Real round():Real max(Real):Real min(Real):Real Además: < > <= >= = <>
Boolean	and or xor not implies if-then-else-endif = <>
String	concat(String):String size():Integer substring(Integer, Integer):String toInteger():Integer toReal():Real toLower() : String toUpper():String Además: = <>

Tabla 10. Tipos básicos de OCL y sus operaciones

2.03 Colecciones

Existe también la posibilidad de manipular colecciones de elementos, para lo que se definen los tipos *Collection, Set, Bag* y *Sequence,* siendo estos tres últimos especializaciones de *Collection* (que, por otra parte, es un tipo abstracto)*:*

- Un *Set* (conjunto) representa un conjunto de elementos en el que puede haber sólo un ejemplar de cada elemento.
- Una *Bag* (bolsa) representa un conjunto de elementos en el que cada elemento puede aparecer más de una vez.
- Una *Sequence* (secuencia) es una *bag* cuyos elementos están ordenados, en el sentido de que los elementos tienen un número de orden que podemos utilizar para acceder a un elemento dado.

Las operaciones definidas para *collection* y heredadas por sus tres especializaciones se muestran en la siguiente tabla:

Operaciones de *Collection*	
Operación	**Tipo del resultado**
size	Integer
includes(x : OclAny)	Boolean
count(x : OclAny)	Integer
includesAll(c : Collection)	Boolean
isEmpty	Boolean
notEmpty	Boolean
sum	Integer o Real
exists(expr:OclExpression)	Boolean
forAll(expr : OclExpression)	Boolean
iterate(expr : OclExpression)	expr.evaluationType

Tabla 11. Operaciones del tipo *Collection,* también disponibles en *Set, Bag* y *Sequence*

2.03.1 *Set* (conjunto)

Además de las operaciones anteriores, *Set* incluye las siguientes:

Operaciones de *Set(T)* (*T* es el tipo de los elementos del conjunto)		
Operación	**Tipo devuelto**	
union(s : Set(T))	Set(T)	
union(b : Bag(T))	Bag(T)	
= (s : Set(T))	Boolean	
intersection(s : Set(T))	Set(T)	
intersection(b : Bag(T))	Bag(T)	
- (s : Set(T))	Set(T)	Devuelve los elementos que están en un conjunto pero no en ambos
including(x : T)	Set(T)	añade *x* al conjunto
excluding(x : T)	Set(T)	elimina *x* del conjunto
symmetricDifference(s : Set(T))	Set(T)	conjunto de elementos que están en uno o en otro conjunto, pero no en ambos
select(expr : OclExpression)	Set(expr.type)	subconjunto de los elementos de *self* para los que *expr* es cierta
reject(expr : OclExpression)	Set(expr.type)	subconjunto de los elementos de *self* para los que *expr* es falsa
collect(expr : OclExpression)	Bag(expr.oclType)	*bag* resultante de aplicar *expr* a todos los elementos de *self*
count(x : T)	Integer	Nº de apariciones de *x* en *self.* Como es un *Set,* el resultado es 0 o 1
asSequence	Sequence(T)	
asBag	Bag(T)	

Tabla 12. Operaciones definidas para el tipo *Set*

2.03.2 *Bag* (bolsa)

En la siguiente tabla aparecen las operaciones definidas para el tipo *Bag:*

Operaciones de *Bag(T)* (*T* es el tipo de los elementos de la bolsa)		
Operación	**Tipo devuelto**	
union(s : Set(T))	Bag(T)	Como puede observarse, unir un *Set* y un *Bag* produce siempre un *Bag*
union(b : Bag(T))	Bag(T)	
= (s : Bag(T))	Boolean	
intersection(s : Set(T))	Set(T)	
intersection(b : Bag(T))	Bag(T)	
including(x : T)	Bag(T)	añade *x* a la bolsa
excluding(x : T)	Bag(T)	elimina todas las apariciones de *x* de la bolsa
select(expr : OclExpression)	Bag(T)	"subbolsa" de los elementos de *self* para los que *expr* es cierta
reject(expr : OclExpression)	Set(T)	"subbolsa" de los elementos de *self* para los que *expr* es falsa
collect(expr : OclExpression)	Bag(expr.oclType)	*bag* resultante de aplicar *expr* a todos los elementos de *self*
count(x : T)	Integer	Nº de apariciones de *x* en *self*. Como es una *Bag*, el resultado es 0 o mayor que cero.
asSequence	Sequence(T)	
asSet	Set(T)	

Tabla 13. Operaciones definidas para el tipo *Bag*

2.03.3 *Sequence* (secuencia)

Para terminar con las colecciones, enumeramos en la siguiente tabla las operaciones del tipo *Sequence:*

Operaciones de *Sequence(T)* (*T* es el tipo de los elementos de la secuencia)		
Operación	**Tipo devuelto**	
count(x : T)	Integer	Nº de apariciones de *x* en *self*.
= (s : Sequence(T))	Boolean	
union(s : Sequence(T))	Sequence(T)	
append(x : T)	Sequence(T)	Añade el elemento al final
preprend (x : T)	Sequence(T)	
subsequence(inf:Integer, sup:Integer)	Sequence(T)	
at(pos : Integer)	T	
first	T	
last	T	
including(x : T)	Sequence(T)	Añade el elemento al final (como append)
excluding(x : T)	Sequence(T)	elimina todas las apariciones de x
select(expr : OclExpression)	Sequence (T)	
reject(expr : OclExpression)	Sequence (T)	
collect(expr : OclExpression)	Sequence (expr.oclType)	
asBag	Bag(T)	
asSet	Set(T)	
iterate(expr : OclExpression)	expr.evaluationType	

Tabla 14. Operaciones definidas para el tipo *Sequence*

2.03.4 Significado de algunas operaciones de las colecciones

No todas las operaciones se comportan igual en todos los tipos de colecciones. Algunas de estas particularidades son las siguientes:

	Significado en...		
Operación	**Set**	**Bag**	**Sequence**
=	Todos los elementos de ambos conjuntos son iguales	Todos los elementos son iguales y, además, cada uno aparece el mismo número de veces en ambos conjuntos	Como en Bag, pero además todos los elementos están ordenados de igual forma
including	Añade un elemento al conjunto si es que no estaba ya presente	Añade el elemento a la bolsa, aunque ya esté presente	Añade el elemento al final de la secuencia
excluding	Elimina el elemento del conjunto	Elimina de la bolsa todas las apariciones del elemento	Elimina de la secuencia todas las apariciones del elemento
intersection	Puede aplicarse para obtener la intersección de dos conjuntos (devuelve un conjunto), dos bolsas (devuelve una bolsa) o un conjunto y una bolsa (devuelve un conjunto)		No aplicable

Tabla 15. Significado de algunas operaciones en las colecciones. La tabla no incluye el tipo *Collection* porque es abstracto

La operación *forAll* equivale al cuantificador universal (∀), y se utiliza para denotar que todos los elementos de una colección deben cumplir cierta condición. Del mismo modo, la operación *exists* representa el cuantificador existencial (∃), y se usa cuando en una colección debe existir al menos un elemento que cumpla la condición.

La operación más compleja pero también más potente para trabajar con colecciones es la operación *iterate*, que se utiliza para recorrer una colección y hacer operaciones con sus elementos. La sintaxis general de *iterate* es la siguiente:

```
colección->iterate(elemento : Tipo1 ;
                   resultado : Tipo2 = expresión
                   | operación(elemento, resultado)
```

Un pseudocódigo en "pseudoJava" que describe el funcionamiento de esta operación es el siguiente:

```
resultado = asignación inicial
for (int i=0; i<colección.size(); i++) {
    elemento=colección.elementAt(i);
    resultado=operación(elemento, resultado);
}
return resul;
```

3. Ejemplos

A continuación, introduciremos la notación de OCL escribiendo algunas restricciones para el diagrama de clases de la siguiente figura:

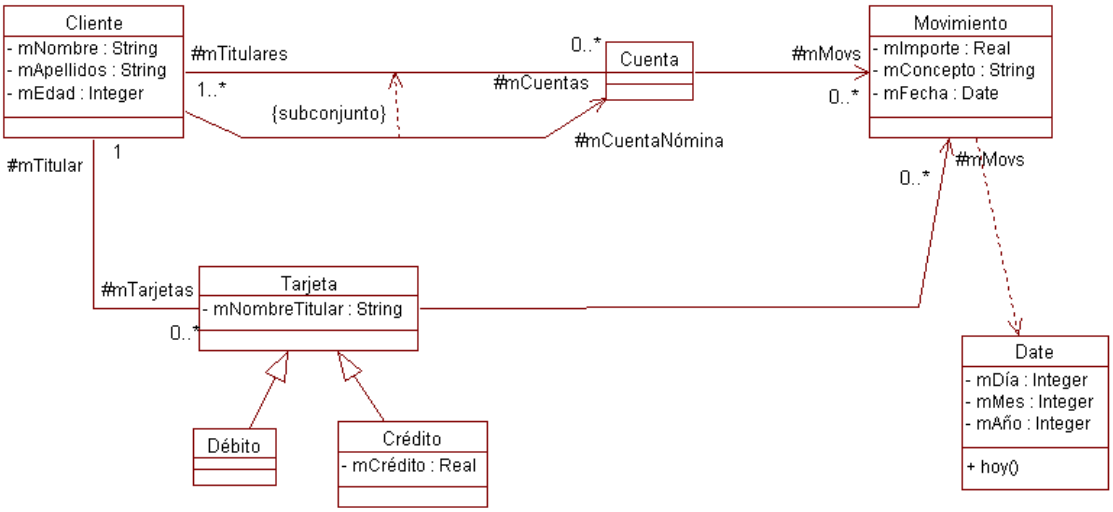

Figura 146. Un diagrama de clases

Queremos representar las siguientes restricciones:

1) La edad de cualquier cliente es siempre mayor o igual a cero.

2) En toda cuenta debe haber al menos un titular de 18 años o más.

3) El saldo de toda cuenta debe ser siempre mayor o igual a cero (el saldo se calcula como la suma del importe de sus movimientos).

4) El titular de toda tarjeta de crédito debe tener la nómina domiciliada en la misma cuenta que está asociada a la tarjeta.

5) El nombre del titular que figura en toda tarjeta es la concatenación del nombre y apellidos del titular.

6) La cantidad gastada en cierto mes con una tarjeta de crédito es menor o igual al crédito de la tarjeta.

Todas estas restricciones son invariantes, ya que son condiciones que deben ser satisfechas en todo momento por las cuentas y tarjetas de crédito. La clase dentro de la cual especificamos la invariante es lo que se llama su *contexto*. Así, la primera invariante, que afecta al atributo *mEdad* de la clase *Cliente,* se expresa en OCL como sigue:

```
context Cliente inv:
 mEdad>=0
```

Restricción 1

La segunda restricción se refiere a los titulares de las cuentas; el contexto, en este caso, es *Cuenta:*

```
context Cuenta inv:
 self.mTitulares->exists(c : Cliente | c.mEdad>=18)
```

Restricción 2

En este segundo caso:

- *self* se refiere a la instancia de *Cuenta* (entiéndase *self* como el *this* en Java o C++);
- *self.mTitulares* representa la colección (*collection*) de titulares de esta *Cuenta:* obsérvese que *mTitulares* es el nombre de rol de la clase *Cliente* en la asociación desde *Cuenta* a *Cliente*; además, esta expresión devuelve una colección porque la multiplicidad de este lado de la asociación es *.
- La flecha se utiliza para representar una operación que se realiza sobre una colección. Como podemos ver en la Tabla 11, *exists* es una operación definida para el tipo *collection.*
- A *exists* le estamos pasando como parámetro una *OclExpression* con la que decimos que, en la colección de titulares de esta cuenta, existe un cliente cuya edad es mayor o igual que 18 años.

La tercera restricción se especifica navegando hacia la relacion Moviento, y utilizando el nombre de rol "mMovs". Luego de navegar la relación podemos obtener el importe de cada moviento haciendo referencia al atributo mImporte.

```
context Cuenta inv:
self.mMovs.mImporte->sum() > 0
```

Restricción 3

Podríamos haber definido una expresión OCL de tipo definición para representar en una variable denominada saldo, cuyo valor es el resultado de la suma de movimientos de una cuenta.

```
context Cuenta
def: saldo : real = self.mMovs.mImporte->sum()
```

Restricción 4

De esta forma la variable *saldo* puede ser utilizada en la definición de una expresión OCL semánticamente equivalente a la definida en la restricción 3.

```
context Cuenta inv:
saldo > 0
```

Restricción 5

Supongamos que la clase *Cuenta* tuviera un atributo derivado, denominado *mSaldo* (es decir, su valor es calculable a partir de otros elementos). En este caso, podemos especificar una regla de derivación.

```
context Cuenta::msaldo
derive : self.mMovs.mImporte->sum()
-- el saldo es igual a la suma de los importes
-- de los movimientos de una cuenta.
```

Restricción 6

La cuarta restricción es más compleja, ya que involucra a *Crédito*, *Cliente* y *Cuenta*. Sin embargo, puede describirse de manera muy sencilla de este modo:

```
context Crédito inv:
 self.mCuentaAsociada.mTitulares->count(self.mTitular)>0
 and
 self.mTitular.mCuentaNomina=self.mCuentaAsociada
```

Restricción 7

También podríamos haber escrito esta expresión de la siguiente forma:

```
context Crédito inv:
 self.mCuentaAsociada.mTitulares->includes(self.mTitular)
 and
 self.mTitular.mCuentaNomina=self.mCuentaAsociada
```

Restricción 8

En la primera parte de esta última restricción estamos diciendo que el número de apariciones del titular de esta tarjeta (*self.mTitular*) en la colección de titulares de la cuenta asociada a esta tarjeta (*self.mCuentaAsociada.mTitulares*) tiene que ser mayor que cero; en la segunda parte decimos que la cuenta nómina del titular de la tarjeta es la misma cuenta asociada a esta tarjeta.

Otra característica interesante de este último ejemplo es que estamos accediendo, desde el contexto *Crédito*, a *Cuenta* y a *Cliente*, que *Crédito* hereda de *Tarjeta.*

La quinta restricción es muy sencilla:

```
context Tarjeta inv:
 self.mNombreTitular=
    mTitular.mNombre.concat(" ").concat(mTitular.mApellidos)
```

Restricción 9

Para describir la sexta restricción debemos seleccionar aquellos movimientos de la tarjeta de crédito que hayan tenido lugar este mes y sumarlos:

```
context Crédito inv:
   self.mCrédito>=self.mMovs->
    select(m:Movimiento|m.mFecha.ObtenerMes()= Dia.ObtenerMes(Dia.hoy()))
     ->collect(mImporte)->sum()
```

Restricción 10

Todas las restricciones que afectan a colecciones pueden expresarse mediante la operación *iterate*. Así, la segunda restricción de la Sección 3 (al menos uno de los titulares de las cuentas debe tener 18 años o más), por ejemplo, puede expresarse así:

```
context Cuenta inv:
   let r: Set(Titulares) = self.mTitulares->Iterate(
               p: Titular ;
               mayores: Set(Titulares) = Set{} |
               if p.mEdad > 18 then
                    mayores.including(p)
                    else
                    mayores
               endif
               ) in
    r.notEmpty()
```

Restricción 11

```
context Cuenta inv:
     let r : integer = self.mTitulares->iterate(
               p : Titular ;
               mayores : integer =0   |
               if p.mEdad>=18 then
                   mayores +1
               else
                   mayores
               endif
              ) in
      r>0
```

Figura 147. La Restricción 2, representada mediante la operación *iterate*

El ejemplo de la Figura 147 está representando el siguiente pseudocódigo (véase página 172):

```
int resultado=0;
for (int i=0; i<this.mTitulares.size(); i++) {
   Titular p=(Titular) mTitulares.elementAt(i);
   if (p.mEdad>=18)
      mayores=mayores+1;
}
return resul;
```

Figura 148. Pseudocódigo correspondiente a la figura anterior

Una restricción adicional que podemos representar con respecto de la Figura 146 es el hecho de que la cuenta con la nomina de un cliente es una de las cuentas de las que dicho cliente es titular. Esto ya está realmente especificado en el diagrama UML, pero puede también describirse en OCL de esta manera:

```
context Cliente inv:
 self.mCuentas->includes(self.mCuentaNómina)
```

Restricción 4. La instancia representada por un rol está incluida en la colección denotada por otro rol

Si la multiplicidad de *mCuentaNómina* fuera *, la restricción sería muy parecida, pero utilizaríamos la operación *includesAll*:

```
context Cliente inv:
 self.mCuentas->includesAll(self.mCuentaNómina)
```

Restricción 5. Las instancias representadas por un rol están incluidas en la colección denotada por otro rol

4. Tipos del modelo

Como se observa, en la Restricción se accede al atributo *mes* de *mFecha* (que, en el diagrama de la Figura 146, es de clase *Date*); también se accede a la operación *hoy()* de *Date*. Esto significa que, dentro de una especificación OCL, podemos utilizar cualquier tipo definido en el modelo UML. En este ejemplo concreto, además, ocurre que la operación *hoy()* es una operación de clase (estática), y hacemos referencia a ella poniendo el nombre de la clase delante del nombre de la operación (como hacemos en Java al ejecutar un método estático).

Por tanto, podemos utilizar tanto los atributos como las operaciones del tipo, y utilizarlos para escribir las restricciones.

5. Precondiciones y postcondiciones

Así como el contexto de una invariante es la clase, tipo o interfaz para cuyas instancias se define la invariante, el contexto de las precondiciones y postcondiciones es la clase que es dueña de la operación en la cual se define la pre/postcondición.

De forma general, la sintaxis para la escritura de pre y postcondiciones es la siguiente:

```
context Tipo :: operación(parámetro1 : Tipo1, ...) : TipoDevuelto
     pre : ...
     post : ...
```

Restricción 6. Sintaxis general de precondiciones y postcondiciones

El *Tipo* que se especifica tras el *context* es la clase, interfaz o tipo al que pertenece la *operación*.

Para denotar cuál debe ser el resultado devuelto por la operación que se está describiendo se utiliza la palabra clave *result*. En las postcondiciones se utiliza *@pre* para denotar el valor de una variable antes de la operación. Así, por ejemplo, una posible descripción de la operación *retirar(importe : real) : real* de la clase *Cuenta* (que toma como parámetro y devuelve la cantidad que se retira) podría ser la siguiente:

```
context Cuenta :: retirar(importe : real) : real
 pre : importe>0
       and getSaldo()>=importe
 post : result=importe
```

Restricción 7. Descripción de la operación de retirada mediante una pre y una postcondición

En la restricción anterior estamos utilizando la operación *getSaldo()* que, evidentemente, no es una operación estándar de OCL; para poder utilizar esta operación en una expresión OCL, debe encontrarse adecuadamente descrita también en OCL. Podemos

basarnos en la Restricción (página 174), en la que describíamos una invariante para un supuesto atributo *mSaldo*, pero adaptándola a la operación *getSaldo*:

```
context Cuenta :: getSaldo() : real :
 post : result = self.mMovs.mImporte->sum
```

Restricción 8. Descripción de la operación *getSaldo*

6. Representación de diagramas de estados

La siguiente figura muestra un posible diagrama de estados que representa parte del comportamiento de una Cuenta.

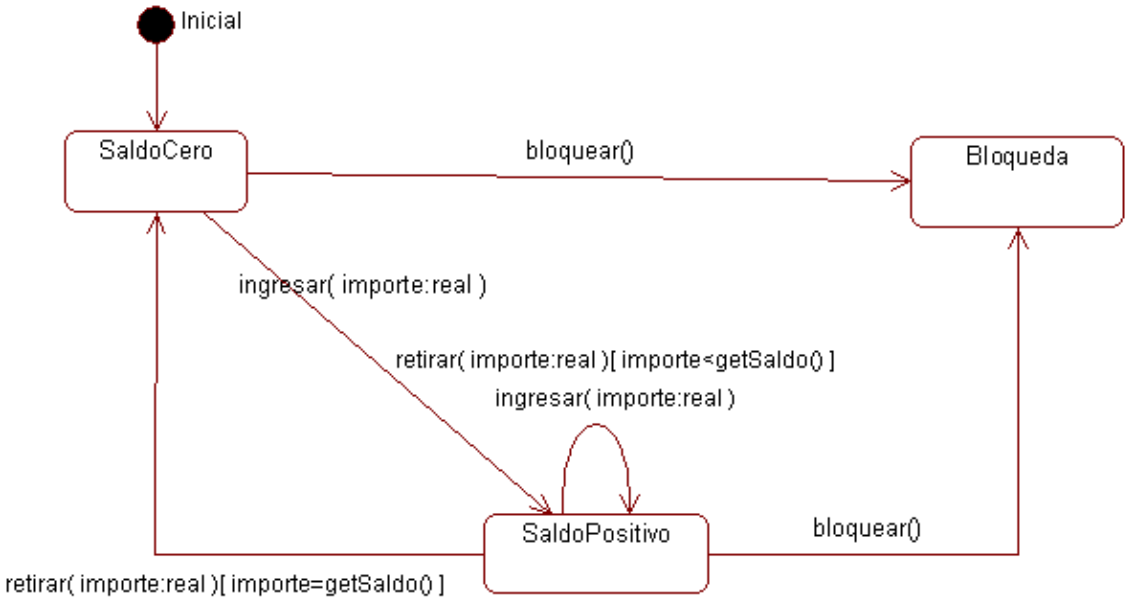

Figura 149. Un diagrama de estados

De forma general, de cada transición podemos considerar que su estado origen y su posible condición ("guarda") son precondiciones de la operación (que corresponde al evento que etiqueta la transición), y que el estado destino es una postcondición. En este ejemplo, la única precondición para que ocurra la operación *ingresar* es que la cuenta tenga o saldo positivo o saldo cero: o, lo que es lo mismo, que no esté bloqueada. Tras la ejecución de la operación, la cuenta tendrá siempre saldo positivo. En otras palabras:

```
context Cuenta :: ingresar(importe : real)
 pre  : not Bloqueada
 post : SaldoPositivo
```

Restricción 9. Representación de parte del diagrama de estados de la Figura 149

La operación *retirar* es ligerísimamente más compleja:

```
context Cuenta :: retirar(importe : real)
 pre  : SaldoPositivo and importe<=getSaldo()
 post : (SaldoPositivo and getSaldo()>0)
        or
       (SaldoCero and getSaldo()=0)
```

Restricción 10. Representación de parte del diagrama de estados de la Figura 149

7. Enumeraciones

Como se observa en las dos restricciones anteriores, consideramos los estados como atributos de tipo booleano. También puede considerarse que cada clase tiene un atributo *state* de tipo *Enumeration*. En OCL, se hace referencia a los posibles valores de una enumeración anteponiendo el carácter # a dicha enumeración. De este modo, las restricciones anteriores pueden representarse así:

```
context Cuenta :: ingresar(importe : real)
 pre  : state<> #Bloqueada
 post : state=#SaldoPositivo
```

Restricción 11. La Restricción 9, considerando ahora un atributo *state* dentro de *Cuenta*

```
context Cuenta :: retirar(importe : real)
 pre  : state=#SaldoPositivo
 post : (state=#SaldoPositivo and getSaldo()>0)
        or
        (state=#SaldoCero and getSaldo()=0)
```

Restricción 12. La Restricción 12, considerando ahora un atributo *state* dentro de *Cuenta*

No obstante, si describimos la *Cuenta* y su estado como hacíamos en la Figura 141, (página 154) en la que utilizábamos el patrón *Estado*, las restricciones se escribirían haciendo referencia al atributo *mEstado* y utilizando, por ejemplo, la operación *oclIsTypeOf* (véase Tabla 8) para obligar a establecer el tipo de *mEstado* a uno de los tipos especializados de *EstadoCuenta:*

```
context Cuenta inv :
 mEstado.oclIsTypeOf(ConDinero) xor
 mEstado.oclIsTypeOf(SinDinero) xor
 mEstado.oclIsTypeOf(Bloqueada)
```

Restricción 13. Una restricción para los estados de la Figura 141

8. Mensajes

En OCL también existe la notación para representar mensajes. Para representar un mensaje enviado desde una instancia a otra se utiliza el operador ^. Así, por ejemplo, para denotar que, tras sacar dinero con una tarjeta de débito se efectúa una retirada en la cuenta asociada (véase Figura 146), podemos escribir lo siguiente:

```
context Débito :: sacarDinero(importe : real)
 pre : mCuentaAsociada.getSaldo()>=importe
 post : mCuentaAsociada^retirar(importe)
```

Restricción 14. Un paso de mensajes

En esta última restricción no sería preciso representar el estado de *mCuentaAsociada* tras la ejecución de la operación, ya que la operación *retirar* en *Cuenta* se encuentra totalmente descrita en OCL.

Por otro lado, en ocasiones se envía una lista de mensajes a un grupo de objetos. Puede hacerse referencia a la lista de mensajes mediante el operador ^^, que devuelve una secuencia de elementos de tipo *OclMessage*. Supongamos que, al eliminar una *Cuenta*, se da valor *false* a un atributo *mBorrado* de todos sus movimientos y que, además, se han enviado tantos mensajes como movimientos haya:

```
context Cuenta :: borrar()
 let mensajesDeBorrar : Sequence(OclMessage) = mMovs^^borrar()
 post : mensajesDeBorrar.size = mMovs.size
```

Restricción 15

En ocasiones no se conocen los valores de los parámetros que se envían con el mensaje, utilizándose en este caso interrogaciones en lugar de nombres de los parámetros.

El valor devuelto por un mensaje enviado es accesible a través de la operación *result(),* cuyo tipo es el mismo que devuelve el mensaje. Supongamos que la operación *sacarDinero* de *Débito* llama a *retirar* de su cuenta asociada, y que devuelve además el saldo que queda en la cuenta tras la retirada:

```
context Cuenta :: retirar(importe : real) : boolean
    pre : importe>=sel.getSaldo()
    post :
       let mensaje : OclMessage = mCuentaAsociada.retirar(importe) in
       mensaje.hasReturned()
       and
       mensaje.result()=mCuentaAsociada.getSaldo()
```

Restricción 16