OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*

P. Sánchez, P. Letelier, I. Ramos, Ó. Pastor

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia. Valencia (España)
{ppalma, letelier, iramos, opastor}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Resumen

OASIS v.3.0 es un enfoque formal para la especificación de modelos conceptuales siguiendo el enfoque orientado a objetos. Esta nueva versión supone diversas mejoras y aportaciones que podemos resumir en: (1) definición de un marco formal homogéneo para el modelo OASIS, (2) incorporación de la perspectiva cliente, (3) consideración de un concepto de proceso más ámplio que engloba obligaciones y prohibiciones, (4) enriquecimiento de los mecanismos de definición de clases complejas (agregación y herencia), (5) incorporación de un metanivel que permite abordar aspectos relativos a evolución y reutilización de especificaciones software, y (6) presentación de OASIS como lenguaje con una sintaxis bien definida.

1 Introducción

OASIS (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [14] es un enfoque formal para la especificación de modelos conceptuales siguiendo el paradigma orientado a objetos. Otras propuestas con una motivación similar a la de OASIS son TROLL [12], CMSL [24], LCM [8], Albert [5], Oblog [21] y Gnome [22].

Este artículo presenta las principales características de OASIS 3.0. El modelo OASIS se expone en [14] como un lenguaje de especificación con una sintaxis textual bien definida. Además se establecen los principios formales del lenguaje y se realiza un recorrido exhaustivo de la capacidad expresiva del mismo.

*Este trabajo está finaciado por el proyecto *MENHIR* de la Comisión Interministerial de Ciencia y Tecnología con referencia TIC97-0593-C05-01.

Con respecto de la versión anterior OASIS 2.2 [18], en OASIS 3.0 se han introducido las siguientes mejoras:

- Establecimiento de un marco formal uniforme que cubre todas las secciones de la especificación de una clase.
- Incorporación de la perspectiva cliente. Además de los servicios que puede proveer un objeto, toda solicitud de servicio hecha por un objeto es tratada también como una acción para sí mismo.
- Redefinición del concepto de proceso, distinguiendo entre procesos de obligación (operaciones) y procesos de prohibición (permisos). Las operaciones permiten modelar un comportamiento algorítmico asociado a un servicio no atómico provisto por el objeto. Los protocolos constituyen una extensión al concepto *process* de la anterior versión.
- Enriquecimiento de los mecanismos usados para definir clases complejas: agregación y herencia. Respecto a la primera se han caracterizado con mayor detalle las propiedades asociadas a los distintos tipos de agregación [6, 7, 11]. En herencia se ha introducido una visión más amplia, incluyendo tres formas de herencia [26]: particiones estáticas, particiones dinámicas y roles.
- Extensión del modelo OASIS con un metanivel que permite abordar de forma homogénea y elegante los aspectos de: evolución, reutilización y gestión de configuraciones de software [3].

2 Introducción a OASIS 3.0

En OASIS, un **objeto** es un proceso observable cuya vida está caracterizada por la ocurrencia de acciones, tanto si son solicitadas como si son recibidas por el objeto. Así, un objeto puede actuar como cliente o como servidor según esté solicitando u ofreciendo servicios[1]. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado. En la vida de un objeto específico, las acciones cuyo cliente es el mismo objeto son acciones solicitadas. Las acciones cuyo servidor es el mismo objeto son acciones servidas.

Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Cada objeto tiene un evento de creación (que inicia su vida) y opcionalmente uno de destrucción. Los eventos pueden ser estructurados como **procesos** en un nivel "molecular". Además de la semántica propia del sublenguaje utilizado para especificar procesos, añadiremos una semántica adicional para distinguir entre procesos de obligación y procesos de prohibición. Un proceso de obligación es un servicio de mayor nivel ofrecido por el objeto. Un caso particular de proceso de obligación es cuando, además, se asume que el proceso actúa como "todo o nada" y se denomina **transacción**. Un proceso de prohibición impide la ejecución de determinadas secuencias de acciones en la vida del objeto definiendo las secuencias que están permitidas.

En OASIS, un sistema de información es entendido como una sociedad de objetos autónomos y concurrentes que interactúan entre ellos mediante acciones. Cada objeto puede solicitar sólo los servicios que le son accesibles de otros objetos. Las **interfaces** son un mecanismo que permite establecer relaciones de accesibilidad entre objetos, definiendo la visión que tiene cada objeto respecto de la sociedad de objetos.

Cada objeto encapsula su estado y las reglas que rigen su comportamiento. Como es habitual en todo entorno OO, los objetos pueden ser vistos desde dos perspectivas distintas: estática y dinámica. Desde el punto de vista estático, llamaremos **atributos** al conjunto de propiedades que describen estructuralmente al objeto. Los valores asociados a cada propiedad estructural del objeto caracterizan el **estado del objeto** en un instante dado. La evolución de los objetos (perspectiva dinámica) viene caracterizada por la noción de **cambio de estado**: la ocurrencia de una acción puede generar cambios (definidos por **evaluaciones** y **derivaciones**) en los valores de atributos. La actividad de un objeto está determinada por un conjunto de reglas: **precondiciones**, **restricciones de integridad**, **disparos**, **protocolos** y **operaciones**.

Por otro lado, la vida de un objeto puede representarse como una secuencia de **pasos**. Cada paso está formado por un conjunto de acciones que ocurren en un instante dado de la **vida del objeto.** Las acciones son abstracciones de cambios atómicos en el estado del objeto.

Cada objeto tiene un identificador único (**oid**) proporcionado implícitamente por el sistema. Sin embargo, el objeto es referenciado mediante mecanismos de identificación pertenecientes al espacio del problema. Una función de identificación establece correspondencias entre los mecanismos de identificación y el *oid* del objeto.

Llamamos **tipo** a la plantilla que describe la estructura y el comportamiento de un objeto. Una **clase** se compone de un nombre de clase, una o más funciones de identificación y un tipo. Una clase compleja es aquella definida utilizando otras clases. Las relaciones entre clases disponibles en OASIS son **agregación** y **herencia**.

3 Semántica de OASIS

La semántica de OASIS es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[2] posibles que un objeto puede alcanzar. Sea F el conjunto de Fórmulas bien formadas (Fbf) evaluadas sobre el estado (mundo asociado) en el cual se encuentra el objeto, $\underline{A}$ el conjunto "ground" de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos instanciados posibles. Las funciones τ y ρ se definen como:

$$\tau : F \to 2^W$$
$$\rho : 2^{\underline{A}} \to (W \to W)$$

La función τ asigna a una fórmula en la lógica de estado (Lógica de Predicados de Primer Orden) el conjunto de mundos en los cuales se satisface. La función ρ asigna a cada paso una relación binaria entre mundos, la cual es la semántica declarativa del lenguaje. Siendo $\mu \in 2^{\underline{A}}$ un paso y $w, w' \in W$, el significado buscado es: $(w, w') \in \rho(\mu)$ si y sólo si

[1] Los conceptos cliente y servidor son usados en su sentido genérico, no se refieren específicamente a arquitecturas distribuidas cliente/servidor. Conceptos cercanos a lo que hemos preferido denominar "cliente" son: agente [5, 9] y actor [25, 1].

[2] De acuerdo con lo dicho, los estados son aserciones (fórmulas), los mundos son estructuras sobre las que dichas fórmulas son interpretadas.

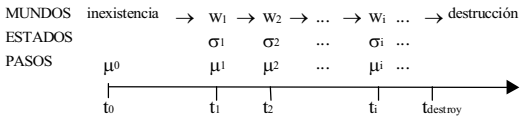

Figura 1: Relación entre mundos, estados y pasos del objeto.

la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

Definición 1 ***Transición válida.*** *Cada par* $(w, w') \in \rho(\mu)$ *se denomina transición válida, con* $w, w' \in W$ *y* $\mu \in 2^{\underline{A}}$.

La relación entre estados, mundos, pasos y la vida del objeto se representa en la Figura 1 donde t_i es el tiempo en el cual se produce el i-ésimo *tic* de reloj para el objeto. Obviamente, se cumple que: $\models_{w_i} \sigma_i \ , i = 1, ..., n$.

De forma global, OASIS está basado en Lógica Deóntica [2]: la lógica de las obligaciones, prohibiciones y permisos. En [14] se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase OASIS y cómo la plantilla completa de la clase se corresponde con fórmulas en una variante de Lógica Dinámica [10] formalizado en [16]. Así, la interpretación de las diferentes fórmulas en Lógica Dinámica, usadas en la plantilla de clase OASIS, es la siguiente:

1. Fórmulas del tipo $\psi \rightarrow [a]\phi$, en particular $[a]\phi$, cuando ψ es *true* (cualquier estado del objeto).

   Siendo $w \in \tau(\psi)$ el mundo actual, si a ocurre entonces el mundo alcanzado es $w' \in \tau(\phi)$. Así, las evaluaciones establecen las propiedades que deben satisfacerse en el mundo alcanzado como efecto atómico de la ejecución de una acción. Dicho de otra forma: $\models_w \psi \wedge \models_{w'} \phi$.

2. Fórmulas de prohibición[3] $\neg\phi \rightarrow [a]false$

   Siendo $w \in \tau(\phi)$ el mundo actual, si a ocurre entonces no existe por definición una transición válida. Es decir, una fórmula de prohibición impide que una acción ocurra estando el objeto en determinados estados: aquellos donde $\models_w \neg\phi$.

3. Fórmulas de obligación $\phi \rightarrow [\neg a]false$

   Siendo $w \in \tau(\phi)$ el mundo actual, si la acción a no ocurre entonces no existe por definición un mundo válido alcanzable. Es decir, una obligación fuerza a que la acción a ocurra cada vez que el estado del objeto es tal que $\models_w \phi$.

[3] Una acción está permitida en un estado si no existe una prohibición que se satisfaga en dicho estado.

4 Especificación de clase

Una especificación OASIS es, esencialmente, un conjunto estructurado de definiciones de clase. Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por las relaciones de clase que la definen.

A continuación mostramos un resumen de los aspectos más significativos de OASIS como lenguaje, recorriendo cada una de las partes que forman la plantilla de una clase OASIS.

4.1 Atributos

Los atributos son propiedades estructurales de las instancias de una clase y poseen un tipo asociado. En OASIS se distinguen tres tipos de atributos: *constantes*, *variables* y *derivados*. Para cada atributo definido existirán funciones y operaciones asociadas, disponibles para ser usadas en la especificación de la plantilla y que son útiles para manipular atributos cuya cardinalidad es mayor que uno. Además se puede especificar la representación escogida para atributos multivaluados (**list**, **set** o **bag)**.

4.2 Restricciones de integridad

Las restricciones de integridad son fórmulas basadas en el estado del objeto y que deben ser satisfechas cada vez que se ejecuta una acción independiente (no incluída en una transacción) o cuando se ejecuta la acción que finaliza una transacción. Se pueden clasificar en *estáticas* o *dinámicas*, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Para especificar restricciones de integridad dinámicas se utilizan los operadores temporales tradicionales.

Ejemplo 1 *Algunas restricciones de integridad dinámicas:*

- `sometimes {saldo>100000} since {interes>10};`

- `always (> 3 months)`

```
{proyecto_terminado=true}
since {proyecto_inscrito=true};
```

4.3 Eventos

Un evento es la abstracción de un cambio de estado atómico e instantáneo que le puede acontecer a un objeto.

- **new** y **destroy** indican que se trata del evento de creación o de destrucción de instancias, respectivamente. Ambos eventos son únicos para cada clase. El evento **new** tiene como parámetros los valores para los atributos constantes y variables del objeto.

- Distinguimos entre eventos privados (sin la etiqueta **calling to members** o **sharing with members**), eventos implicados (con la etiqueta **calling to members**) y eventos compartidos (con la etiqueta **sharing with members**). Los primeros son servicios ofrecidos o requeridos por un objeto y cuya ocurrencia depende sólo del comportamiento del objeto en sí mismo. Los eventos implicados y los compartidos corresponden a servicios provistos por un objeto agregado, coordinados con servicios en sus objetos componentes. En el primer caso la ocurrencia del evento en el objeto agregado implica necesariamente la ocurrencia del correspondiente servicio en sus objetos componentes. Cuando se trata de eventos compartidos, dicha implicación es en ambos sentidos.

Ejemplo 2 *La clase* ***préstamo****, definida como agregación entre un objeto de la clase* ***libro*** *y uno de la clase* ***socio****, tiene su evento de creación compartido con eventos en sus componentes. Considerando que* ***codigo*** *y* ***numero*** *son los mecanismos de identificación de* ***libro*** *y* ***socio****, respectivamente, en la especificación de la clase* ***préstamo*** *tendremos:*

```
class prestamo
    ...
events
prestar(Fecha:date,NumSocio:nat,NumLibro:nat)

new sharing with members
 libro(codigo,[NumLibro]).ser_prestado(Fecha),

 socio(numero,[NumSocio]).obtener_libro;
    ...
end class
```

4.4 Evaluaciones

Las evaluaciones son fórmulas en lógica dinámica que establecen cómo las acciones afectan el estado del objeto. Los atributos variables modifican sus valores según lo establecido en las evaluaciones. Estas fórmulas dinámicas para evaluaciones son del tipo $\psi \rightarrow [a]\phi$, y se interpretan como: "si en un determinado estado del objeto se satisface ψ y ocurre la acción a, en el estado inmediatamente posterior del objeto se satisface ϕ". Es decir, la condición de evaluación ψ se evalúa y se satisface en el estado en el que ocurre la acción y ϕ se satisface en el instante inmediatamente posterior a la ocurrencia de a.

Ejemplo 3 *Una evaluación que establezca una cierta relación entre* ***deposito(Cantidad:nat)*** *y* ***reintegro(Cantidad:nat)*** *respecto del atributo* ***saldo*** *de una cuenta bancaria.*

```
[deposito(Cantidad)]saldo:=saldo+Cantidad;
[reintegro(Cantidad)]saldo:=saldo-Cantidad;
```

4.5 Precondiciones

En algunas ocasiones no es suficiente la iniciativa de un cliente para que una acción ocurra en el servidor sino que además ciertas condiciones deben satisfacerse en dicho servidor. Si la precondición asociada a una acción a no se satisface en el servidor, en lugar de acontecerle la acción a le acontece la acción $\neg a$ (es decir, cualquier otra acción, incluso ninguna).

En el contexto de lógica dinámica, las precondiciones tienen la forma $\neg\phi\,[a]\,false$, donde ϕ es una Fbf interpretada como una condición para la ocurrencia de la acción indicada. El significado es "si ϕ no se cumple, la ocurrencia de la acción no lleva al objeto a un siguiente estado".

Ejemplo 4 *El evento* ***reintegro*** *en la clase* ***cuenta*** *tiene la siguiente precondición:*

```
reintegro(Cantidad) if {saldo >= Cantidad};
```

4.6 Disparos

Un disparo modela la siguiente situación: si un objeto está en un estado que satisface una condición de disparo entonces debe generarse la obligación asociada. Si el objeto en el cual se establece la obligación es precisamente el cliente de dicha acción, dicha acción ocurre como una solicitud de servicio desde dicho objeto. Si el objeto en el cual existe la

obligación sólo es el servidor de dicha acción, deberá esperar hasta que el cliente de la acción le solicite el servicio de la acción[4].

Para caracterizar la actividad declarada en el párrafo de disparos del lenguaje OASIS utilizamos la fórmula dinámica $\phi[\neg a]false$, cuyo significado es: "si ϕ se satisface y no se ejecuta la acción indicada, entonces el objeto no alcanza un estado válido. En otras palabras, si se cumple ϕ, la acción debe ocurrir para que el objeto alcance un estado siguiente". Consideraremos que se cumple:

- Los disparos, por representar una obligación, deben ser consistentes con los permisos definidos mediante precondiciones y protocolos.
- El disparo establecido por una obligación se mantiene mientras la condición de disparo se satisface, es decir, el objeto continuará solicitando un determinado servicio o esperando por ofrecer cierto servicio, según corresponda.

Ejemplo 5 *Cuando el porcentaje de utilización del disco de un servidor llega al 80%, el objeto de la clase* ***servidor*** *solicita a los usuarios que liberen espacio de disco.*

```
usuario(everyone)::
aviso(fecha,hora,``sin espacio'')
    when {porcentaje_utilizado > 80
        and avisado_80_%=false};
```

4.7 Operaciones y protocolos

Existen tres semánticas posibles que pueden ser asociadas en la especificación de un proceso en OASIS:

(a) La semántica del lenguaje utilizado para su especificación. En OASIS, el lenguaje utilizado para especificar procesos es un subconjunto de CCS [17], compartiendo así una semántica basada en la noción de sistema de transición etiquetado y todas las propiedades formales establecidas en CCS.

(b) La semántica de permisos, es decir, una especificación de proceso establece secuencias de acciones que *pueden* ocurrir.

(c) La semántica de obligaciones, es decir, una especificación de proceso establece secuencias de acciones que *deben* ocurrir.

[4]Esto corresponde al concepto de *action waiting* que se trata en [14] en el capítulo de interacción entre objetos.

Un proceso en OASIS tendrá la semántica (a) junto con la semántica (b) ó (c) dependiendo de si se trata de un protocolo o una operación, respectivamente. Así, la sección *operations* se utiliza para especificar secuencias obligatorias de acciones y la sección *protocols* para especificar secuencias permitidas de acciones. Un caso particular de proceso de obligación es el que actúa como una transacción, es decir, con la política de "todo o nada". En este caso se puede declarar con el calificativo de *transaction* para el proceso. Asumiremos siempre por defecto que el proceso no actúa como transacción.

Existen dos secciones en las cuales se describen especificaciones de proceso: *operations* y *protocols*. Para cada operación o protocolo debe existir un estado que representa el estado de inicio del proceso.

Ejemplo 6 *Un protocolo en la clase* ***máquina****. Se trata de una máquina de chocolatinas que acumula hasta tres monedas como crédito*[5]*.*

```
maq:
  maq   =  moneda.MAQ1;
  MAQ1  =  chocolatina.maq+moneda.MAQ2+
           ::return_moneda.maq[6];
  MAQ2  =  chocolatina.MAQ1+ moneda.MAQ3+
           ::return_moneda.MAQ1;
  MAQ3  =  chocolatina.MAQ2 +
           ::return_moneda.MAQ2;
```

4.8 Clases Complejas

Una clase compleja utiliza en su definición otras clases (simples o complejas). Las relaciones para la definición de clases complejas son agregación y herencia. Consideraremos que se cumple:

- Independientemente de las relaciones entre clases complejas establecidas, la especificación de una clase se hace separadamente de las relaciones entre clases en las cuales pueda participar.
- Diremos que una clase compleja tiene propiedades emergentes si tiene propiedades adicionales a las capturadas implícitamente por la relación

[5]En OASIS, para evitar detallar cada elemento de la tupla $< Cliente, Servidor, Servicio >$ que representa a una acción, se adoptan las siguientes simplicaciones: a) Si el servidor es el propio objeto, se escribe *Cliente:Servicio* y si el cliente no es relevante sólo se escribe *Servicio*. b) Si el cliente es el propio objeto, se escribe *Servidor::Servicio*. *c)* En el caso particular de comunicación *self*, la acción de requerir se escribe *::Servicio* y la acción de proveer *self:Servicio* (o simplemente *Servicio*).

[6]Permiso para realizar un *action sending*, en el cual tanto el cliente como el servidor son el mismo objeto.

entre clases que la define (caso contrario, la especificación explícita de su plantilla de clase es opcional).

4.8.1 Agregación

Una agregación modela la noción de relaciones estructurales entre objetos. Un caso particular es cuando la agregación representa la noción *parte_ de*. Una asociación (clase cuyas instancias son un grupos de objetos) se representa mediante una agregación multivaluada definida con un sólo componente. Dado que no puede existir comunicación por *action sharing* o *action calling* fuera de la estructura de agregación, el conjunto total de instancias que un objeto de una clase agregada puede referenciar de alguna de sus componentes está restringido a las instancias que forman parte del objeto agregado. Este aspecto es remarcado por la etiqueta **members** en la definición de eventos de la clase agregada.

Ejemplo 7 *Consideremos el objeto agregado* `coche` *cuyos componentes son* `motor`, `ruedas` *(debe tener 4 ruedas y opcionalmente una de repuesto) y el* `chasis` *que representa la carcasa. Esto nos lleva a tener la clase* `coche` *como objeto agregado con componentes* `chasis`, `ruedas` *y* `motor`*:*

```
coche aggregation of
static inclusive chasis
    towards(1,1) from(1,1);
static inclusive motor
    towards(1,1) from(0,1);
dynamic inclusive rueda
    towards(4,5) from(0,1)
list[‘‘adelante izq’’,‘‘adelante der’’,
‘‘atras izq’’,‘‘atras der’’,‘‘repuesto’’];
```

Un `chasis` no puede existir si no está asociado a un `coche` y un `coche` tiene exactamente un `chasis`. Por ser **static** `chasis` forma parte del `coche` de manera permanente y el que sea **inclusive** conlleva que cualquier cosa que se quiera hacer con el `chasis` es a través del `coche`. El `motor` puede existir independientemente de su relación con un `coche`. En cuanto a las `ruedas` se refiere, vemos que pueden existir fuera del contexto de un `coche` pero, sólo pueden formar parte de un `coche`. La pertenencia de una `rueda` a un `coche` puede variar con el tiempo (**dynamic**). La palabra reservada **towards** representa las cardinalidades vistas desde el objeto agregado hacia los componentes. La palabra **from** representa las cardinalidades vistas desde un objeto del componente hacia el agregado.

La asociación es vista como una agregación en la cual sólo existe un componente. Al realizarse la agrupación de instancias en la asociación, la especificación **where** permite seleccionar los objetos de la clase del componente que satisfacen la fórmula especificada. La palabra reservada **grouping by** indica que cada instancia del componente (grupo de objetos) se crea (o destruye) automáticamente agrupando los objetos de la clase componente según los valores de dichos atributos en los objetos de la clase del componente.

Visibilidad desde el agregado al componente Independientemente del tipo de agregación, existe visibilidad de atributos del componente desde el agregado. La coordinación entre actualizaciones de los atributos en el objeto componente respecto de actualizaciones en el objeto agregado se realiza implícitamente mediante eventos compartidos. En lo que a acciones se refiere, si la agregación es inclusiva todas las acciones del componente son también acciones en el agregado. Adicionalmente, usando **calling to members** se pueden definir eventos en el agregado que se corresponden con eventos en los componentes. Si la agregación es relacional cualquier coordinación entre ocurrencia de acciones en el agregado y en el componente debe establecerse mediante especificaciones **calling to members** o **sharing with members**, explícitamente declaradas en la clase agregada.

Para cada componente y atributo definido (con cardinalidad mayor que uno) existirán funciones y operaciones asociadas disponibles para ser utilizadas en las fórmulas de la plantilla. Algunas de estas funciones son **count**, **min**, **sum, position**, **first,** etc.

Ejemplo 8 *El siguiente ejemplo redefine la clase* `coche` *y establece relaciones entre los objetos agregados y sus componentes.*

```
class coche
...
derived attributes
    temperatura_del_coche:int;
    presion_promedio:real;
    gastos_1998:int;
derivations
    temperatura_del_coche:=
        motor.temperatura;
    presion_promedio:=
        avg(rueda.presion)
            where {rueda.instalada=true};
    gastos_1998:=
        sum(reparacion.importe)
          where {año=1998};
...
events
```

```
    arrancar_coche alias for
        motor.encender;
  borrar_reparaciones_un_año(ValorAño:nat)
    calling to members
    reparacion(ref,[ValorAño,_]).eliminar;
  girar_izq
    calling to members
        rueda[‘‘adelante izq’’].girar_izq,
        rueda[‘‘adelante der’’].girar_izq;
  girar_der
    calling to members
        rueda[‘‘adelante izq’’].girar_der,
        rueda[‘‘adelante der’’].girar_der;
  ...
  end class

  class reparacion
  identification
    ref:(año, mes)
  constant attributes
    año:nat; mes:nat;
    importe:real;
  events
    introducir new;
    eliminar destroy;
  ...
  end class



  coche aggregation of
  static inclusive chasis
        towards(1,1) from(1,1);
  static inclusive motor
        towards(1,1) from(0,1);
  dynamic relational reparacion
        towards(0,*) from(1,1);
  dynamic inclusive rueda
        towards(4,5) from(0,1)
  list[‘‘adelante izq’’, ‘‘adelante der’’,
    ‘‘atras izq’’, ‘‘atras
der’’,‘‘repuesto’’];
```

Ejemplo 9 *Dada la siguiente definición para un atributo:*

```
  números:nat list[1..10];.
```

Se podrían definir los siguientes atributos derivados usando la signatura implícita para dicho atributo:

```
  derivations
    sin_numeros:= {count(numeros)=0};
    tiene_numeros_grandes:=
    {(count(numeros)
        where value>100) > 0};
    suma_de_numeros:= sum(numeros);
    maximo_numero:= max(numeros);
    primer_numero:= first(numeros);
```

4.8.2 Herencia

Mediante la herencia podremos especializar (o generalizar) propiedades definidas en las clases. Utilizaremos las particiones estáticas en aquellas especializaciones que son fijas desde la creación del objeto. Por otro lado, definiremos particiones dinámicas para expresar que la partición a la que pertenezca el objeto no es fija y que depende de la ocurrencia de acciones específicas o de los valores de los atributos en un determinado instante. Los cambios de una instancia entre particiones dinámicas corresponden con lo que se denomina **proceso de migración**. Además de las particiones, utilizaremos las clases de rol cuando un objeto pueda desempeñar temporalmente el comportamiento especificado en la clase de rol. En el caso de roles, a diferencia de las particiones dinámicas, un objeto puede desempeñar varios roles de la misma clase de rol y de forma simultánea. Por esto, el objeto rol es un objeto distinto con su propio *oid* pero cuya existencia está supeditada a la existencia del objeto base.

Las acciones de creación de instancias son siempre dirigidas a las *especies*. Una especie es una clase como producto cartesiano entre las particiones de más bajo nivel en la misma jerarquía de clases. Consideramos que existe compatibilidad de comportamiento para las particiones estáticas y dinámicas. En cambio, para los roles consideramos como mínimo compatibilidad de signatura con extensiones tanto horizontales como verticales [23]. Además se considera que la especificación de una clase se hace separadamente de sus relaciones de herencia existentes. Cada especie involucra a varias clases por lo que conlleva la herencia múltiple de las propiedades especificadas individualmente. Para aquellas clases especies que incorporen propiedades emergentes (además de las implícitas por la herencia múltiple) se incluirá explícitamente la plantilla de clase correspondiente.

Ejemplo 10 *La clase (especie)* `camión*diesel` *puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Para ello, especificaremos la plantilla de la clase* `camión*diesel` *como para el resto de clases.*

Considerando la compatibilidad de comportamiento en las particiones estáticas y dinámicas, deberán cumplirse ciertas restricciones para la especificación de precondiciones, protocolos, evaluaciones, disparos, operaciones y restricciones de integridad en las subclases definidas.

4.8.3 Particiones estáticas

Las instancias de las subclases definidas por particiones estáticas están asociadas a una partición a partir de su creación y se mantienen en ella durante toda su vida.

Ejemplo 11 *Dos particiones estáticas de una misma clase:*

```
camion, coche, otro_vehiculo
    static specialization of vehiculo;
gasolina, diesel, otro_tipo
    static specialization of vehiculo;
```

Las propiedades de la superclase y de las subclases indicadas en estas jerarquías se definirán separadamente mediante plantillas de clase (cuando sea necesario).

4.8.4 Particiones dinámicas

Un proceso migratorio es aquel por el que una instancia pasar de estar especializada en una subclase de una partición a otra subclase de la misma partición. Se dispone de dos formas alternativas para especificar el proceso de migración: por la ocurrencia de ciertas acciones en determinados estados del objeto, y por partición de los posibles estados del objeto en función de los valores que presenten sus atributos.

Particiones dinámicas por la ocurrencia de acciones específicas Cada partición dinámica expresa las distintas subclases de las que puede formar parte un objeto perteneciente a la clase que se particiona. El poder expresar dicho proceso de migración mediante la ocurrencia de acciones aporta al lenguaje una riqueza expresiva considerable. Las acciones implicadas en el proceso migratorio pertenecen a la subclase de la partición que se abandona. Por defecto, el **new** de las instancias es el servicio en la acción indicada al comienzo del proceso de migración.

Ejemplo 12 *Una especialización dinámica de la clase* `coche` *determinada por la ocurrencia de* `crear_coche`*,* `reparar` *y* `estropear` *puede ser:*

```
funcionando, estropeado
    dynamic specialization of coche
migration relation is
    coche = crear_coche.funcionando;
    funcionando = estropear.estropeado;
    estropeado = reparar.funcionando;
```

Según lo dicho, la creación de una instancia de `coche` implica comenzar perteneciendo a la partición `funcionando`. Cuando sea una instancia de `funcionando` le ocurrirán acciones de la signatura de `coche` más, posiblemente, otras de la subclase `funcionando`. Entre ellas está la acción `estropear` que pertenece a la signatura de `funcionando`. Su ocurrencia implica salir del proceso que representa a `funcionando` y entrar en el indicado en el proceso migratorio, esto es, la subclase `estropeado`.

Partición dinámica en función de valores de atributos Dado que en este tipo de partición el proceso migratorio se define en función el estado de los objetos, cuando el objeto alcanza un nuevo estado puede implicar su migración de una subclase a otra en la partición. La modificación del estado del objeto determina el proceso de migración entre las subclases de la partición dinámica. Además, la partición que se haga del conjunto de estados a partir de los valores de los atributos debe ser completa y disjunta.

Ejemplo 13 *Una partición dinámica de la clase* `cuenta` *en función del atributo* `saldo` *puede ser:*

```
no_rentable where {saldo<100000},
medio_rentable where {saldo>=100000
        and saldo<1000000},
muy_rentable where {saldo>=1000000}
        dynamic specialization of cuenta;
```

Una acción de creación debe especificar en qué partición estática se sitúa la nueva instancia. La subclase de la partición dinámica se determina implícitamente a partir del valor del atributo `saldo` incluido en el proceso de creación.

4.8.5 Roles

Mediante roles representamos conductas distintas de un objeto. Una clase puede tener asociada diferentes subclases de rol, cada una representando un patrón de conducta específico para un objeto en dicha subclase. El objeto sigue siendo instancia de una sola clase pero puede desempeñar distintos roles a lo largo de su existencia. Las instancias de una subclase de rol pueden ser creadas y destruidas. Un objeto puede estar asociado al mismo tiempo a dos o más instancias de roles, pudiendo desempeñar varias instancias de la misma subclase de rol simultáneamente.

Ejemplo 14 *Un ejemplo de roles definidos sobre la clase* `persona`*.*

```
estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

Se ha querido representar con las cardinalidades el que una instancia de la clase `persona` puede desempeñar, simultáneamente, como máximo `10` *roles* de `empleado` o, alternativamente, como máximo un rol de `estudiante`. También es posible que una instancia de `persona` no desempeñe ningún rol.

El evento de creación es enviado a la clase `persona`. Posteriormente, si ocurre el evento `ser_matriculado` (que representa el **new** de `estudiante`), entonces la `persona` pasa a desempeñar el rol de `estudiante`. Si destruimos el objeto de `persona`, automáticamente se destruye el objeto `estudiante`, lo contrario no se cumple.

Ejemplo 15 *Consideremos la partición dinámica de* ***persona*** *junto a los roles definidos sobre la misma clase en el ejemplo anterior:*

```
niño where {edad<14},
adolescente where {14<=edad and edad<18},
adulto where {18<=edad}
    dynamic specialization of persona;

estudiante towards(0,1) ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

En este caso, la existencia del rol no supone aumentar el número de especies dado que es un nuevo objeto el que se crea cuando se comienza a desempeñar el rol. En cambio, a futuras particiones (estáticas o dinámicas) de la clase de rol sí podrían dar lugar a nuevas especies.

4.8.6 Creación y destrucción de instancias

En las particiones estáticas, dinámicas y roles, cada subclase hereda el conjunto de mecanismos de identificación definidos en sus superclases. Siempre existirá la posibilidad de definir mecanismos de identificación a los por defecto heredados. La creación de un nuevo objeto debe hacerse dentro de la especie a la que va a pertenecer. La acción de creación va dirigida a la especie como intersección de todas las clases en las que el nuevo objeto se especializa. Se consideran los siguientes casos con vistas a la creación de objetos:

1: La especie es intersección únicamente de subclases de particiones estáticas. En este caso, se deben especificar todas las subclases involucradas dado que quien crea el objeto posee toda la información necesaria para determinar la especialización concreta. Las acciones de creación deben dirigirse a las especies completas, es decir a `camión*diesel, coche*gasolina`, etc.

2: La especie es intersección de subclases de especializaciones dinámicas y opcionalmente estáticas. En este caso, se deben especificar todas las subclases estáticas involucradas si bien en las dinámicas distinguimos dos situaciones:

   – Aquellas subclases de especializaciones dinámicas cuyo proceso migratorio está definido sobre atributos. En este caso el modelo automáticamente deduce en qué subclase de dicha partición dinámica se especializa.

   – Aquellas subclases de especializaciones dinámicas cuyo proceso migratorio está definido sobre las acciones de migración. En este caso si sólo existe un punto de entrada en el proceso migratorio, la acción de creación será la indicada como comienzo del proceso y el modelo lo deduce automáticamente. En cambio, si el proceso de migración tiene varios puntos de entrada, se deberá explicitar la clase de comienzo, y de la misma forma que antes, la acción de creación será la incluida en la subclase dinámica indicada.

Dado que se pueden definir especializaciones estáticas y dinámicas de las clases de rol, la acción de creación del rol involucra también la creación del objeto en alguna de las especies de intersección aplicándose el mecanismo descrito anteriormente.

4.9 Interfaces

Una interfaz es una clase virtual, esto es, una vista de una clase definida por una proyección de la plantilla de una clase. Esta nueva plantilla establece las propiedades de los objetos de la clase original (servidores) que pueden ser usadas por determinados objetos clientes. Así, una interfaz también establece una relación de acceso o visibilidad entre clientes y servidores. Las interfaces permiten al diseñador proteger a una clase de usos no deseados, establecer visibilidad ajustada a las necesidades de cada objeto cliente, y establecer relaciones de acceso con el entorno del sistema.

Las interfaces definidas deben ser consistentes con las relaciones de acceso establecidas en la plantilla de cada clase al especificar acciones. Un objeto

cliente no puede solicitar un servicio a un determinado objeto servidor si no existe la correspondiente interfaz que establezca dicha relación de acceso al servicio.

Toda solicitud de servicio desde un cliente a un determinado servidor, especificada en una plantilla de clase, debe poderse deducir desde alguna definición de interfaz. El conjunto de servicios que un cliente puede requerir se obtiene inspeccionando todas las interfaces que se aplican al objeto como cliente. Del mismo modo, los clientes potenciales que pueden requerir un determinado servicio se obtienen inspeccionando todas las interfaces en las cuales se da acceso a dicho servicio.

La interfaz puede establecer relaciones de acceso entre objetos y/o conjuntos de objetos, dependiendo de que la referencia es a un objeto individual, grupo de objetos de una clase o todos los objetos de una clase.

Ejemplo 16 *Una interfaz para el usuario identificado por el valor en* `nombre=''Juan Pérez''`*, encargado de modificar el precio de los artículos del inventario.*

```
interface usuario(nombre,[''Juan Perez''])
            with articulo(someone)
    attributes(all)
    services(cambiar_precio)
end interface
```

5 Conclusiones

Este artículo presenta la versión 3.0 del lenguaje OASIS en el que se ha conseguido integrar tanto los aspectos formales que justifican el modelo como una sintaxis BNF precisa del mismo. Este trabajo se enmarca dentro del proyecto MENHIR en el cual destacamos las líneas de trabajo: (1) animación de especificaciones OASIS usando Redes de Petri Orientadas a Objetos (véanse [19, 20]) y Programación Lógica Concurrente (véanse [13, 15]) y, (2) la definición e implementación de un entorno de especificación y evolución de esquemas conceptuales (véase [4] donde se presenta la interfaz gráfica de usuario de una herramienta que utiliza la metaclase como núcleo de la misma y que permite la definición, validación y evolución de los esquemas conceptuales).

Referencias

[1] Agha G.A. *ACTORS: A model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[2] Åqvist L. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pages 605-714. Reidel, 1984.

[3] Carsí J.A., Ramos I., Penadés M.C., Pelechano V. *Descripción del modelo de objetos de OASIS a través de la Transaction Frame Logic*. I Jornadas de Trabajo en Ingeniería del Software, Sevilla, 1996.

[4] Carsí J.A., Camilleri S., Canós J.H., Ramos I. *Propuesta de Interfaz Gráfica de Usuario homogénea para el Diseño y la Explotación de Sistemas de Información*. JIS'98, III Jornadas de Trabajo en Ingeniería del Software, Murcia, Noviembre, 1998.

[5] Dubois E., Du Bois P., Petit M. *O-O Requirements Analysis: an agent perspective*. In Proc. of the 7th.European Conference on Object Oriented Programming- ECOOP 93, pages 458-481, 1993.

[6] Ehrich H.-D., Goguen J.A., Sernadas A. *A Categorical Theory of Objects as Observed Processes*. LNCS 489, 1990.

[7] Ehrich H.-D. *Objects Specification*. Abteilung Datenbanken, Technische Universität, Braunschweig, Germany, 1995.

[8] Feenstra R.B., Wieringa R.J. *LCM 3.0: a languaje for describing conceptual models*. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[9] Genesereth M.R., Ketchpel S.P. *Software Agents*. Communications of the ACM, 37(7): pages 48-53, 1994.

[10] Harel D. *Dynamic Logic*. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pages 497-694, Reidel 1984.

[11] Hartmann T., Junglaus R., Saake G. *Aggregation in a Behaviour Oriented Object Model*. Abt. Datenbanken, TU Braunschweig, 1992.

[12] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*.

ACM Transactions on Information Systems, Volume 14, Number 2, pages 175-211, 1995.

[13] Letelier P., Sánchez P., Ramos I. *Animation of systems specifications using concurrent logic programming.* Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97, Aix-en-Provence, France, 1997.

[14] Letelier P., Ramos I., Sánchez P., Pastor O. *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto.* Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, Universidad Politécnica de Valencia, 1998. `www.dsic.upv.es/users/oom/books.html`.

[15] Letelier P., Sánchez P., Ramos I. *Prototyping a requirements specification through an automatically generated concurrent logic program.* First International Workshop on Practical Aspects of Declarative Languages (PADL'99), Texas, USA, Enero 1999.

[16] Meyer J.-J.Ch. *A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic.* In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.

[17] Milner R. *Communication and Concurrency.* Prentice Hall Series in Computer Science, C.A.R. Hoare, Series Editor, 1989.

[18] Pastor O., Ramos I. *OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[19] Sánchez P., Letelier P., Ramos I. *Constructs for prototyping information systems with Object Petri Nets.* Proceeding of the IEEE International Conference on SMC'97, pages 4260-4265, Orlando, 1997.

[20] Sánchez P., Letelier P., Ramos I. *Una Aproximación a la representación de OASIS gráfico en Redes de Petri Orientadas a Objeto.* III Jornadas de Ingeniería del Software (JIS'98), Noviembre, Murcia, España, 1998. (aceptado)

[21] Sernadas C., Gouveia P., Sernadas A. *Oblog: Object-Oriented, logic-based conceptual modelling.* Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1992.

[22] Sernadas C., Ramos J. *Gnome: Sintaxe, semântica e cálculo.* Research Report, Instituto Superior Técnico, Secção de Ciêcia da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1994.

[23] Wegner P., Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* LNCS 322, ECOOP pages 55-77, 1988.

[24] Wieringa R.J. *A conceptual model specification language (CMSL Version 2).* Vrije U., The Netherlands, 1992.

[25] Wieringa R.J., Meyer J.-J.Ch. *Actors, Actions and Initiative in Normative System Specification.* Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.

[26] Wieringa R., Jonge W., Spruit P. *Roles and dynamic subclasses: a modal logic approach.* Vrije U., The Netherlands, 1995.