Modelado Conceptual con un Lenguaje Formal y Orientado a Objeto*

Pedro Sánchez **Patricio Letelier** **Isidro Ramos** **Oscar Pastor**

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

Palabras Clave: Ingeniería de Requisitos, Modelado Conceptual, Modelado Orientado a Objeto

Resumen

El modelado conceptual tiene por objetivo el establecer los requisitos funcionales del sistema. El modelo conceptual establece el vínculo entre el espacio del problema y el espacio de la solución, constituyendo la base para las tareas posteriores en el desarrollo del sistema de información. Los enfoques más populares para abordar el modelado conceptual están basados en notaciones informales, donde la verificación y validación suelen ser poco exhaustivas y difíciles de realizar. Los métodos formales ofrecen una mejora en estos aspectos pero su introducción en entornos industriales de desarrollo de software no se ha popularizado por carencia de métodos y herramientas adecuadas. $\mathcal{OASIS}$ es un enfoque formal para el modelado conceptual orientado a objeto. En $\mathcal{OASIS}$, desde sus primeras versiones se ha puesto especial hincapié en su utilización como modelo subyacente en métodos y herramientas de aplicabilidad industrial. Así, siguiendo el modelo $\mathcal{OASIS}$, se han desarrollado una serie de herramientas integradas en un entorno CASE y un método ad hoc denominado OO-METHOD. Este artículo presenta algunas de las principales características incluidas en la última versión de $\mathcal{OASIS}$. Se describen brevemente los conceptos generales del modelo $\mathcal{OASIS}$ y se recorren algunos aspectos del lenguaje de especificación asociado.

1 Introducción

Generalmente, los requisitos del usuario son representados usando modelos semiformales. Notacionalmente los métodos orientados a objeto más populares han sido fundidos en UML (*Unified Modeling Language*) [12]. En los métodos semiformales la verificación y validación suele ser poco exhaustiva, obteniendo finalmente una implementación en un lenguaje imperativo sobre el cual es muy difícil estudiar su comportamiento y establecer su corrección. Muchos fracasos en proyectos de software han sido atribuidos a problemas en ingeniería de requisitos, tales como: requisitos mal documentados, requisitos imposibles de satisfacer o requisitos que no corresponden a las necesidades del usuario. En este sentido, y particularmente en el modelado conceptual, es interesante el uso de métodos formales.

*Este trabajo está financiado por el proyecto *MENHIR* (Métodos, Entornos y Herramientas para Ingeniería de Requisitos), de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01.

OASIS (*Open and Active Specification of Information Systems*) [5] es un enfoque formal para la especificación de modelos conceptuales siguiendo el paradigma orientado a objetos. Las propuestas formales más cercanas a *OASIS* son: TROLL [4], LCM [3], ALBERT [2] y OBLOG [14]. En términos generales, a diferencia de otros enfoques, en *OASIS* desde sus primeras versiones se ha puesto especial interés en tres aspectos esenciales que facilitan su integración en entornos industriales de desarrollo de software:

- Establecimiento de un método de desarrollo ad hoc para *OASIS*, llamado OO-METHOD [10].
- Generación automática de código a partir de especificaciones *OASIS*, orientada a la validación [6, 7] y construcción incremental del producto final [11].
- Construcción de herramientas que soporten el proceso de modelado conceptual. Se ha construido un entorno CASE basado en la versión precedente de *OASIS* y adaptado a OO-METHOD.

En este aspecto en OBLOG también el esfuerzo se han orientado hacia herramientas y un soporte metodológico que permita su uso industrial. Las otras propuestas se han mantenido principalmente en el ámbito académico.

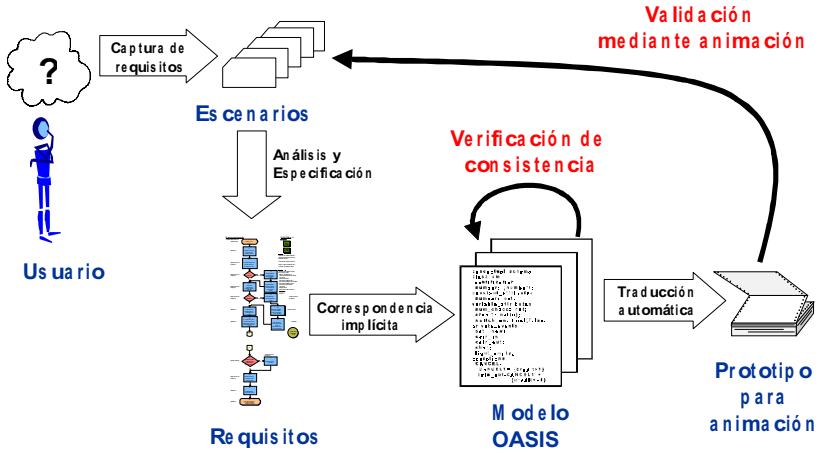

Figura 1: Modelado Conceptual basado en OASIS

La Figura 1 ilustra un proceso de modelado conceptual basado en *OASIS*. La captura de requisitos se realiza mediante técnicas de especificación de escenarios que permitan su comparación total o parcial respecto al comportamiento exhibido por un prototipo del sistema. A partir de los escenarios establecidos se obtienen los elementos de especificación que forman el modelo conceptual. La especificación se realiza mediante técnicas textuales y gráficas que de forma implícita determinan el modelo *OASIS* del sistema. El modelo conceptual es verificado según las reglas de construcción de *OASIS*. Mediante generación automática se obtiene un prototipo con funcionalidad equivalente a la especificación *OASIS*. A través de animación del prototipo la validación se orienta a la satisfacción de los requisitos expresados en los escenarios.

El objetivo de este trabajo es presentar las principales características de *OASIS* 3.0, la última versión publicada de *OASIS*. El resto del trabajo está organizado como se detalla a continuación. En la sección 2 se introducen los conceptos básicos de *OASIS*. Posteriormente,

la sección 3 resume los aspectos formales de $\mathcal{OASIS}$. En la sección 4 se presentan los apartados de especificación en una plantilla de clase. El capítulo 5 describe las relaciones entre clases. Finalmente, en el capítulo 6 se exponen las conclusiones.

2 Conceptos Básicos de $\mathcal{OASIS}$

En $\mathcal{OASIS}$, un **objeto** es un proceso observable cuya vida está caracterizada por la ocurrencia de acciones, que corresponden a servicios requeridos o provistos por el objeto, es decir, un objeto puede actuar como cliente o como servidor. 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. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado.

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.

Los eventos pueden ser estructurados como **procesos** en un nivel "molecular". Además de la semántica propia del sublenguaje utilizado para especificar procesos (un sublenguaje de CCS [9]), se añade una semántica adicional para distinguir entre procesos de obligación (**operación**) y procesos de prohibición (**protocolo**). Una operación es un servicio de mayor nivel ofrecido por el objeto. Un caso particular de operación es cuando, además, se asume que el proceso actúa como "todo o nada" y se denomina **transacción**. Una operación determina secuencias obligadas de acciones en la vida del objeto, en cambio, un protocolo establece secuencias permitidas.

Cada objeto encapsula su estado y las reglas que rigen su comportamiento. Como es habitual en un entorno orientado a objeto, 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 determinan 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 en los valores de atributos (definidos por **evaluaciones** y **derivaciones**). La actividad de un objeto está determinada por un conjunto de fórmulas: **precondiciones**, **restricciones de integridad**, **disparos**, **protocolos** y **operaciones**.

Cada objeto tiene un identificador único (**oid**) proporcionado implícitamente por el sistema. Sin embargo, la referencia a un objeto se hace 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, uno o más mecanismos de identificación y un tipo. Una clase compleja es aquella definida utilizando otras clases. Las relaciones entre clases disponibles en $\mathcal{OASIS}$ son **agregación** y **especialización**.

3 Semántica de $\mathcal{OASIS}$

La semántica de $\mathcal{OASIS}$ es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[1] 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

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

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:

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

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

En [5] se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase $\mathcal{OASIS}$ y cómo la plantilla completa de la clase se corresponde con fórmulas en una variante de Lógica Dinámica formalizada en [8].

4 Especificación de clase

Una especificación $\mathcal{OASIS}$ 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 $\mathcal{OASIS}$ como lenguaje, recorriendo algunas de las partes que forman la plantilla de una clase $\mathcal{OASIS}$.

4.1 Atributos

Los atributos son propiedades estructurales de las instancias de una clase. Un atributo es un par <nombre, *sort* de evaluación>, siendo nombre el identificador del atributo y *sort* de evaluación el nombre del conjunto soporte (*carrier*) o valores que puede tomar dicho atributo. En $\mathcal{OASIS}$ se distinguen tres variedades de atributos: *constantes*, *variables* y *derivados*. Para cada atributo existen funciones y operaciones disponibles para ser usadas en la especificación de la plantilla, de acuerdo al tipo abstracto de datos asociado al atributo. Además, para atributos multivaluados se puede especificar la representación escogida utilizando los tipos abstractos de datos predefinidos **list**, **set** o **bag**.

4.2 Eventos

Un evento es la abstracción de un cambio de estado atómico e instantáneo que le puede acontecer a un objeto. Los eventos **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 1 *La clase* ***préstamo****, definida como agregación entre un objeto de la clase* ***libro*** *y uno de la clase* ***socio****, tiene eventos implicados respecto del evento de creación. 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) new calling to members
        libro.ser_prestado(Fecha),
        socio.obtener_libro;
    ...
end class
```

4.3 Evaluaciones

Las evaluaciones son fórmulas en Lógica Dinámica que establecen cómo las acciones afectan el estado del objeto. Estas fórmulas 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 2 *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;
```

4.4 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 se dice que le acontece la acción $\neg a$.

En el contexto de Lógica Dinámica, las precondiciones tienen la forma $\neg\phi \rightarrow [a]\, false$, donde ϕ es una Fbf interpretada como una condición para la ocurrencia de la acción indicada. El significado es "si ϕ no se cumple, después de la ocurrencia de la acción no existe un estado siguiente válido".

Ejemplo 3 *El evento* ***reintegro*** *en la clase* ***cuenta*** *tiene la siguiente precondición:*

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

4.5 Disparos

Un disparo modela la siguiente situación: si un objeto está en un estado que satisface cierta condición entonces la ocurrencia de una determinada acción está obligada. 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.

Un disparo es modelado por la fórmula $\phi \rightarrow [\neg a] false$, 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".

Ejemplo 4 *Cuando el porcentaje de utilización del disco 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};
```

4.6 Operaciones y protocolos

Existen tres semánticas posibles que pueden ser asociadas en la especificación de un proceso en $\mathcal{OASIS}$:

(a) La semántica del lenguaje utilizado para su especificación. En $\mathcal{OASIS}$, el lenguaje utilizado para especificar procesos es un subconjunto de CCS, 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.

Un proceso en $\mathcal{OASIS}$ 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.

Con la sintaxis y semántica establecida para las especificaciones de proceso existe un correspondencia directa de ellas con conjuntos de fórmulas en la Lógica Dinámica utilizada. Es decir, cualquier especificación de procesos puede ser interpretada como fórmulas de cambio de estado ($\psi \rightarrow [a]\phi$) junto a fórmulas de prohibición ($\neg\phi \rightarrow [a]\, false$) cuando se trata de un protocolo o junto a fórmulas de obligación ($\phi \rightarrow [\neg a] false$) cuando es una operación.

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

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

5 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 especialización. Independientemente de las relaciones entre clases 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 entre clases que la define (en caso contrario, la especificación explícita de su plantilla de clase es opcional).

5.1 Agregación

Una agregación modela la noción de relaciones estructurales entre objetos. Dado que no puede existir comunicación por *action sharing* o *action calling* fuera de la estructura de agregación, las instancias a quienes un objeto de una clase agregada puede hacer referencia como componentes está restringido a las instancias componentes que efectivamente forman parte del objeto agregado. Este aspecto es remarcado por la etiqueta **members** en la definición de eventos de la clase agregada.

Ejemplo 6 *Consideremos el objeto agregado* `coche` *cuyos componentes son* `motor`, `ruedas` *(debe tener* `4` *ruedas y opcionalmente una de repuesto) y un* `chasis`*. 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 la multiplicidad vista desde el objeto agregado hacia los componentes. La palabra **from** representa la multiplicidad vista desde un objeto del componente hacia el agregado.

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.

Para cada componente y atributo definido (con multiplicidad 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 7 *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 8 *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);
```

5.2 Especialización

Mediante especialización las propiedades definidas en las clases pueden ser refinadas. La clase original se denomina superclase, las clases obtenidas por especialización se llaman subclases. En $\mathcal{OASIS}$ existen tres mecanismos de especialización: particiones estáticas, particiones dinámicas y grupos de rol. La especialización incorporada en $\mathcal{OASIS}$ está inspirada en los trabajos de Wieringa [16]. La especialización establece jerarquías de clases. Una clase puede ser superclase de más de una jerarquía a la vez. En $\mathcal{OASIS}$ cada jerarquía de especialización es disjunta, es decir, las instancias de la superclase están asociadas a lo más a una subclase. Además, en el caso de particiones la jerarquía de especialización es total, es decir, cada instancia de la superclase es a la vez instancia de una de las subclases.

5.2.1 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 existencia.

Ejemplo 9 *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).

5.2.2 Particiones dinámicas

Un proceso migratorio es aquel por el cual una instancia pasa de una subclase a otra. 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 En este caso el proceso migratorio se define mediante una especificación de proceso que utiliza la misma sintaxis de las operaciones o protocolos. Las acciones incluidas en la especificación de proceso pertenecen a alguna de las acciones de la subclase desde donde se migra. las constantes agentes que definen el proceso se corresponden con los nombres de las subclases de la partición. Por defecto, el **new** de las instancias es el servicio en la acción indicada al comienzo del proceso de migración.

Ejemplo 10 *Una especialización dinámica de la clase* `coche` *determinada por la ocurrencia de las acciones* `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`.

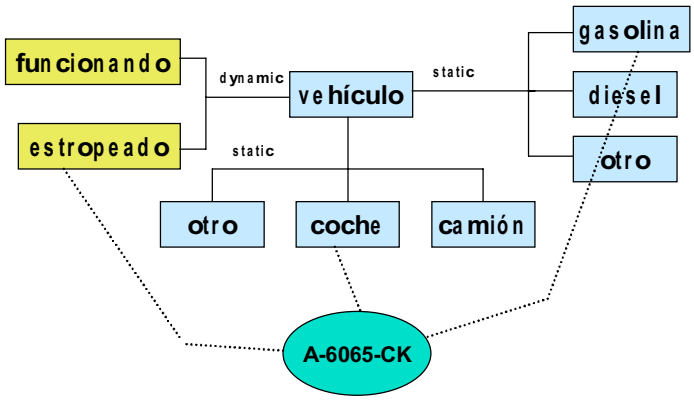

Figura 2: Particiones estáticas y dinámica sobre la clase vehículo.

Partición dinámica en función de valores de atributos En este caso el proceso migratorio queda determinado por el estado de cada objeto. Cada vez que el objeto alcanza un nuevo estado esto puede implicar su migración de una subclase a otra en la partición.

Ejemplo 11 *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;
```

En el ejemplo, la subclase de la partición dinámica a la cual pertenece un objeto de la clase cuenta se determina implícitamente a partir del valor del atributo `saldo`.

Tanto en particiones estáticas como dinámicas cada objeto es instancia a la vez de la superclase y de una (y sólo una) de las subclases, es decir, se trata del mismo objeto (mismo *oid*). Esto tiene las siguientes consecuencias semánticas:

- Debe existir compatibilidad de comportamiento [15] viendo al objeto como instancia de la subclase respecto a su visión desde la superclase.
- Cuando el objeto es destruido en la superclase es destruido también en la subclase y viceversa.

5.2.3 Especies

Una especie es una clase obtenida como producto cartesiano entre las subclases de más bajo nivel en la misma jerarquía. Las acciones de creación de instancias en particiones son siempre dirigidas a las especies. Cada especie incluye las propiedades de 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 12 *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.*

La Figura 2 ilustra las particiones que se han definido en los ejemplos anteriores. Además se muestra cómo un objeto (cuyo valor de mecanismo de identificación es la matrícula A-6065-CK) es instancia de una subclase de cada una de las particiones, es decir, en este caso es instancia de la especie `estropeado*coche*gasolina`.

5.2.4 Grupos de Rol

Mediante grupos de roles representamos conductas distintas de un objeto. Una superclase de un grupo de rol puede tiene asociada subclases de rol, cada una representando un patrón de conducta específico para objetos en dicha subclase. El objeto instancia de la superclase se denomina *player*. El objeto *player* es un objeto diferente al objeto de la subclase de rol (tienen distinto *oid*). Esto tiene tres consecuencias semánticas destacables:

- Un objeto *player* puede estar asociado al mismo tiempo a dos o más instancias de una subclase de rol.
- Pueda no existir compatibilidad de comportamiento entre ambos objetos, es decir, existe mayor libertar en cuando al refinamiento especificado en la plantilla de la subclase.
- El objeto *player* puede seguir existiendo cuando las instancias de una subclase de rol asociadas son destruidas.

Ejemplo 13 *Un ejemplo de un grupo de rol definidos sobre la clase* `persona`*.*

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

Con la restricción de multiplicidad se ha querido representar 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 un rol de `estudiante`. Si destruimos el objeto de `persona`, automáticamente se destruye el objeto `estudiante`, lo contrario no se cumple.

Ejemplo 14 *Consideremos la partición dinámica de* `persona` *junto a las subclases de rol definidas 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;
```

La especificación de un grupo de rol no supone aumentar el número de especies dado que cada subclase de rol constituye la raíz de una nueva jerarquía (pues se trata de objetos distintos). En cambio, las particiones (estáticas o dinámicas) de una subclase de rol dan lugar a nuevas especies.

6 Conclusiones

Los métodos semiformales usados para la especificación de requisitos funcionales continúan sin satisfacer las expectativas de quienes los utilizan. Aunque se agradece el esfuerzo realizado al integrar las principales propuestas de modelado en UML como notación estándar, no existen aportes significativos respecto del proceso de desarrollo ni de la calidad del modelo conceptual obtenido. $\mathcal{OASIS}$ es un enfoque formal donde cada constructor un significado preciso, los conceptos forman un conjunto reducido y consistente, y se ofrece la capacidad expresiva necesaria para el modelado conceptual. Mediante la utilización de herramientas basadas en $\mathcal{OASIS}$ se obtienen las siguiente mejoras: el disponer de una semántica bien definida evita ambigüedades en la especificación, es posible realizar verificaciones más exhaustivas de acuerdo a las propiedades del modelo y se pueden establecer con mayor precisión patrones de implementación para la generación automática de código. Las líneas de investigación actuales en torno a $\mathcal{OASIS}$ incluyen las siguientes áreas: aspectos metodológicos que definen el proceso de desarrollo de software [10], animación automática de especificaciones para la validación temprana de requisitos [6, 7, 13], evolución de esquemas conceptuales integrada en el mismo marco de $\mathcal{OASIS}$ [1] y generación automática de código en entornos de software industriales [11]. Todos estos esfuerzos están dentro del marco de extensión de la herramienta CASE basada en $\mathcal{OASIS}$.

Referencias

[1] 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.* Actas de las III Jornadas de Trabajo en Ingeniería del Software, JIS'98, páginas 141-153, Murcia, 1998.

[2] 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.

[3] 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.

[4] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems.* ACM Transactions on Information Systems, vol.14, num.2, pages 175-211, 1995.

[5] Letelier P., Ramos I., Sánchez P., Pastor O. $\mathcal{OASIS}$ *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. `http://www.dsic.upv.es/users/oom/books.html`.

[6] Letelier P., Sánchez P., Ramos I. *Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program.* Gupta (Ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, LNCS 1551, pages 31-45, Springer-Verlag, 1998.

[7] Letelier P., Sánchez P., Ramos I.. *Un ambiente para Especificación Incremental y Validación de Modelos Conceptuales.* Actas de las Segundas Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software, IDEAS'99, páginas 216-228, Costa Rica, 1999.

[8] 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.

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

[10] Pastor O., Insfrán E., Pelechano V., Romero J., Merseguer J. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods.* Conference on Advanced Information Systems Engineering, CAiSE '97, pages 145-158, Barcelona 1997.

[11] Pastor O., Pelechano V., Insfrán E., Gómez J. *From Object-Oriented Conceptual Modeling to Automated Programming in Java.* Proceedings of 17th International Conference on Conceptual Modeling, ER'98, pages 183-196, Singapure, November 1998.

[12] Rational Software Corporation. *UML notation guide. Versión 1.1*, September 1997. `http://www.rational.com/uml`

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

[14] OBLOG Software S.A. *The OBLOG Software Development Approach (White Paper).* 1999, `http://www.oblog.pt/Download/Documentation.exe`

[15] Wegner P., Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* In Proc. of the 2th.European Conference on Object Oriented Programming, ECOOP'88, LNCS 322, pages 55-77, 1988.

[16] Wieringa R., Jonge W., Spruit P. *Using Dynamic Classes and Role Classes to Model Object Migration.* Theory and Practice of Object Systems, vol.1, num.1, pages 61-83, 1995.