OASIS Versión 3.0
Un enfoque formal para el modelado conceptual orientado a objeto

Patricio Letelier Torres
Pedro Sánchez Palma
Isidro Ramos Salavert
Oscar Pastor López

Abstract

Este documento presenta OASIS 3.0, un enfoque formal para el modelado conceptual de sistemas de información usando el paradigma orientado a objeto. La visión aplicada de OASIS viene dada por un lenguaje de especificación textual. No se pretende promover el uso directo del lenguaje de especificación, sino que éste sirva de representación de más bajo nivel para herramientas gráficas. Mediante una interfaz adecuada se pueden establecer automáticamente correspondencias con los elementos del modelo subyacente. Este trabajo se ha desarrollado en el marco del proyecto MENHIR (Métodos, Entornos y Herramientas para la Ingeniería de Requisitos), parcialmente financiado por la "Comisión Interministerial de Ciencia y Tecnología" (CICYT) y con referencia TIC 97-0593-C05-01. Dicho proyecto está coordinado por el grupo de investigación Modelización Conceptual Orientada a Objeto y Bases de Datos en el DSIC-UPV. Además participan las Universidades de Granada, Murcia, Sevilla y Valladolid.

Agradecimientos. Queremos agradecer a todos los miembros del grupo de investigación de Modelado Orientado a Objeto perteneciente al DSIC-UPV por la discusión generada en nuestras reuniones, en especial al profesor Dr. José Hilario Canós por sus valiosos comentarios. Agradecer también por la revisión del documento a los profesores Juan Sánchez, M. Carmen Penadés, Emilio Insfran y Vicente Pelechano y a los becarios pertenecientes al grupo: José Miguel Barberá y Juan Carlos Molina.

Índice general

Índice de figuras

Capítulo 1

Introducción

La Ingeniería de Requisitos es un campo de reconocida importancia en el ámbito teórico y práctico de la ingeniería del software. Las mayores deficiencias (y desafíos) en el proceso de desarrollo de software se encuentran en sus primeras fases. La distancia entre el espacio del problema (Universo de Discurso) y el espacio de la solución (producto software) hace necesario que la especificación de requisitos del sistema, resultante del proceso de *modelado conceptual*, tenga dos importantes propiedades: debe ser abstracta y declarativa.

El modelo conceptual debe ser abstracto para facilitar la captación y modelado de aspectos del problema de una forma lo más cercana posible a los conceptos del dominio del problema. Esto es muy importante para la validación del modelo obtenido, facilitando la participación del usuario en dicha tarea. En este sentido, el enfoque orientado a objeto (OO) posibilita la elaboración de entornos de producción de software que resuelven problemas clásicos asociados a la ya familiar noción de "Crisis del Software".

Entre las ventajas más destacables del paradigma objetual en este contexto cabe señalar las siguientes:

- La encapsulación bajo el concepto de objeto incluyendo las perspectivas estática y dinámica del sistema estudiado.
- La desaparición de barreras estrictas entre las distintas fases del ciclo de vida o proceso de producción de software.
- La proximidad de sus nociones a los mecanismos cognitivos humanos reduciendo así la distancia entre el problema y su solución.

Por lo anterior, creemos que la orientación a objetos es un enfoque apropiado para el modelado de requisitos de sistemas de información. El modelo que proponemos está basado en dicho enfoque.

Por otra parte, el modelo conceptual debe ser declarativo de manera que permita postergar decisiones de implementación. Esta característica nos separará de lo que denominamos enfoques orientados a objetos clásicos, en los que la especificación es principalmente imperativa y se limita en general a la estructura de los objetos y los perfiles de las operaciones.

La tarea de verificación del modelo exige disponer de algún formalismo preciso y a la vez con toda la capacidad expresiva necesaria para capturar todos los aspectos de interés del problema. Las notaciones gráficas son atractivas pues facilitan el modelado y la legibilidad de una especificación de requisitos, pero pierden estas ventajas cuando se sobrecargan con detalles. Por otra parte, las representaciones puramente textuales, y más aún siendo formales, tienen el grado de detalle que se desee pero exigen más esfuerzo en su utilización. Esto sugiere que metodológicamente la mejor solución es una combinación de ambas técnicas: gráficas y textuales. Como el objetivo es obtener un modelo completo y preciso del sistema, una representación textual final es la más adecuada, pues tanto las técnicas textuales como las gráficas pueden ser finalmente trasladadas a texto para describir el modelo en su totalidad.

OASIS (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [22][23] es un enfoque formal para especificación de modelos conceptuales siguiendo el enfoque orientado a objetos. OO-METHOD [24] es la propuesta metodológica basada en OASIS. OO-METHOD cubre las fases de análisis, diseño e implementación de un sistema de información desde la perspectiva del paradigma de la programación automática, es decir, haciendo énfasis en la animación automática de la especificación, prototipación automática y generación parcial del producto final. Particularmente, en la fase de modelado conceptual, OO-METHOD provee herramientas gráficas para hacer más fácil la construcción del modelo. La especificación OASIS para un sistema es construido implícitamente mediante las herramientas provistas por OO-METHOD.

Lenguajes con una motivación similar a la de OASIS son TROLL [16], CMSL [32], LCM [11], Albert [8], Oblog [29] y Gnome [30].

Este libro presenta OASIS 3.0. El modelo OASIS será expuesto como un lenguaje de especificación con una sintaxis textual bien definida. Se establecerán los principios formales del lenguaje y se realizará un recorrido exhaustivo de su capacidad expresiva.

1.1. Historia

El entorno OO formal y declarativo representado por OASIS y su extensión metodológica (OO-METHOD) constituyen el resultado del intenso trabajo de investigación realizado en el grupo de Modelado Orientado a Objeto en el DSIC-UPV durante los últimos años. Los objetivos de este trabajo de investigación son básicamente:

- Definir un marco formal adecuado para dar cuenta del modelo OO.
- Elaborar un lenguaje que sea lo suficientemente expresivo para ser empleado como herramienta de modelado conceptual de sistemas de información.
- Desarrollar una metodología y herramientas asociadas que permitan el aprendizaje adecuado y uso del modelo propuesto.

El entorno OASIS en su primera versión [22] fue formalizado a través de Teorías de Primer Orden que evolucionan con el tiempo. Este enfoque lo situó históricamente en un entorno de Programación Lógica complementado con un monitor para la gestión de la perspectiva dinámica del sistema. A partir de la segunda versión [23] se cambió de marco formal y se trabajó en el contexto de una variante de Lógica Dinámica que permite representar los operadores de obligación, prohibición y permiso usados en Lógica Deóntica. En este marco, OASIS tiene una base formal más adecuada desde el punto de vista de su comprensión, manteniendo sus buenas propiedades.

Con respecto de la versión anterior OASIS 2.2, 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 y procesos de prohibición. 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. En herencia se ha introducido una visión más amplia, incluyendo tres formas de herencia: 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.

1.2. Modelado orientado a objeto

En OASIS, un **objeto** es un proceso observable [26] 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. Un caso particular es el de acciones cuyo cliente y servidor es el mismo objeto. Esto sucede cuando el objeto se solicita un servicio a sí mismo.

Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Cada objeto tiene un evento de creación (que inicia su

[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 [8][12] y actor [33][1].

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. Con este concepto es posible modelar la noción tradicional de algoritmo. Un caso particular de proceso de obligación es cuando, además, se asume que el proceso actúa como "todo o nada" y lo denominaremos **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[2] 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**.

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. Operacionalmente hablando, la ocurrencia de acciones está asociada a la ejecución de los cambios que pueden producir dichas acciones en el estado del objeto.

Cada objeto tiene un identificador único (**oid**) proporcionado implíci-

[2]Este tipo de concurrencia la denominaremos inter-objetual y permitirá modelar objetos con la capacidad de ser clientes autónomos.

tamente 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.

Llamaremos **tipo** a la plantilla que describe la estructura y el comportamiento de un objeto. Una **clase** se compone de un nombre de clase, una función de identificación y un tipo.

Pueden definirse nuevas clases reutilizando clases ya definidas. Una clase compleja es aquella definida utilizando otras clases. Los operadores entre clases disponibles en OASIS son **agregación** y **herencia**.

1.3. Estructura del trabajo

El trabajo está organizado globalmente en dos partes:

- El modelo OASIS, presentando en detalle su formalización y semántica.
- El lenguaje de especificación OASIS, incluyendo sintaxis, observaciones y ejemplos de su utilización.

La primera parte comprende los capítulos 2, 3, 4 y 5. El capítulo 2 presenta los conceptos básicos del modelo OASIS describiendo la plantilla de clase y cómo ésta puede ser vista como un conjunto de fórmulas en Lógica Dinámica. En el capítulo 3 se establecen los operadores para definir clases complejas basadas en agregación y herencia. El capítulo 4 trata la interacción entre objetos. El capítulo 5 presenta el concepto de metaclase incorporado en el modelo.

La segunda parte consta de dos capítulos. El capítulo 6 hace un recorrido exhaustivo por los distintos bloques de especificación provistos por el lenguaje. En el capítulo 7 se presenta un ejemplo de aplicación del modelo OASIS en el modelado conceptual de un sistema de información.

Adicionalmente, el capítulo 8 contiene las conclusiones de este trabajo. Posteriormente, se presenta la bibliografía utilizada.

Por último, en el anexo A se proporciona un glosario de conceptos utilizados a lo largo del documento y en el anexo B se presenta la sintaxis completa del lenguaje, reuniendo todos los elementos tratados en el capítulo 6.

Parte I

Semántica y formalización de OASIS

Capítulo 2

Clases y Objetos

Una especificación OASIS es, esencialmente, un conjunto estructurado de definiciones de clase. Las clases pueden ser simples o complejas. Una clase compleja es aquella que en su definición utiliza la definición de otras clases (simples o complejas). Las clases complejas se definen utilizando operadores de clase. Estos son agregación y herencia.

Una clase se compone de un nombre de clase, uno o más mecanismos de identificación para instancias de la clase (objetos) y un tipo o plantilla que comparten todas las instancias[1]. La plantilla recoge las propiedades de estructura y de comportamiento compartidas por todos los objetos potenciales de la clase considerada. 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 los operadores de clase que la definen.

Este capítulo formaliza la plantilla de clase utilizada por OASIS. Se muestra cómo una plantilla de clase se corresponde con un conjunto de fórmulas en una variante de Lógica Dinámica, cuya semántica declarativa está definida en el marco de una estructura de Kripke[2].

[1] En el capítulo 5 se aporta la visión "meta" del concepto de clase.

[2] En 1965, Saul Kripke publicó una interpretación semántica de una lógica basada en ciertas álgebras como corolario de su semántica para lógicas modales. La idea central de la semántica de Kripke es caracterizar el valor de verdad de las sentencias mediante estados temporales o estados de conocimiento. Así, una sentencia no es sólo cierta, sino que lo es en cierto estado o en un estado de conocimiento. Estos estados de conocimiento son lo que denominaremos mundos.

2.1. Definiciones preliminares

Definición 1 ***Atributo***. *Un atributo es un par* $\langle$*nombre, sort de evaluación*$\rangle$*, siendo nombre el identificador del atributo y sort de evaluación un conjunto cuyo conjunto soporte (carrier) son los valores que puede tomar dicho atributo.*

Definición 2 ***Atributo evaluado***. *Un atributo evaluado es un par* $\langle nombre,valor\rangle$*, siendo nombre el identificador del atributo y* $valor \in carrier(sort(nombre))$*. Si el valor de un atributo no puede cambiar durante la vida del objeto diremos que el atributo es constante, en caso contrario el atributo es variable.*

Definición 3 ***Servicio***. *Un servicio es un evento o una operación. En el primer caso se trata de la abstracción de un cambio de estado, atómico e instantáneo. Una operación es un servicio no atómico y que, en general, tiene duración.*

Definición 4 ***Acción***. *Una acción es una tupla* $\langle Cliente,Servidor,Servicio\rangle$*, que representa tanto la acción asociada a requerir el servicio en el objeto cliente como la acción asociada a proveer el servicio en el objeto servidor.*

2.2. Plantilla o tipo

Definición 5 ***Plantilla o tipo***. *Una plantilla de clase está representada por una tupla* $\langle Atributos, Eventos, Fórmulas, Procesos\rangle$*.*

- *Atributos* es el alfabeto de atributos evaluados.
- *Eventos* es el alfabeto genérico de eventos. $\forall e \in Eventos$ podemos obtener $\underline{e} = \theta e$ siendo θ una substitución básica de los posibles parámetros del evento.
- *Fórmulas* es un conjunto de fórmulas en diversas lógicas según sea la sección del lenguaje donde se utilizan y que se describen más adelante.

- *Procesos* es el conjunto de especificaciones de proceso. Cada proceso se especifica usando un cálculo de procesos basado en un subconjunto del lenguaje propuesto en CCS[21]. Las especificaciones de proceso se clasifican en operaciones y protocolos.

Para cada clase, asumiremos la existencia implícita de un conjunto A de acciones obtenido a partir de los servicios que los objetos de la clase pueden requerir (cuando son clientes) o proveer (cuando son servidores). Al igual que para los eventos, $\forall a \in Acciones$ podemos obtener $\underline{a} = \theta a$ siendo θ una substitución básica de los posibles cliente, servidor y servicio.

La Lógica de Predicados de Primer Orden es usada en varias secciones del lenguaje, por esto, comenzaremos definiendo las Fórmulas bien formadas (Fbf) en dicha lógica.

Términos

- Cada constante perteneciente a un determinado *sort* es un término de dicho *sort*.

- Cada variable es un término de su *sort* de definición.

- Si $f : s_1 \times s_2 \times ... \times s_n \rightarrow s$ es una función de aridad n, y $t_1, t_2, ..., t_n$ son términos, donde cada t_i pertenece al *sort* s_i, entonces $f(t_1, t_2, ..., t_n)$ es un término del *sort* s.

- Cada atributo de la plantilla de una clase es un término de su *sort* de evaluación.

- Sólo son términos aquellos construidos siguiendo las reglas anteriores.

Átomos

- Las constantes *true* y *false* son átomos.

- Si t_1 y t_2 son términos del mismo *sort*, y R es uno de los siguientes operadores relacionales: $=$, $<>$, $<$, $\leqslant$, $>$ ó $\geqslant$ (asumidos definidos para el *sort* considerado) entonces $t_1\ R\ t_2$ es un átomo.

- Sólo son átomos aquellos construidos siguiendo las reglas anteriores.

Fórmulas bien formadas

- Cada átomo es una Fbf.
- Si ϕ y ϕ' son Fbf, entonces *not* ϕ, ϕ *and* ϕ', ϕ *or* ϕ', son también Fbf con el significado lógico usual.
- Sólo son Fbf aquellas construidas siguiendo las reglas anteriores.

2.3. Ciclo de vida de un objeto

Un objeto tiene dos aspectos: *estructura*, determinada por sus atributos, y *comportamiento*, aspecto asociado a las acciones que le acontecen.

Definición 6 ***Estado del objeto****. Llamaremos estado del objeto al conjunto de sus atributos evaluados.*

Ejemplo 1 *El estado de un objeto de la clase* `válvula` *podría ser:*

```
{<abierta,true>, <caudal,50>}
```

Ejemplo 2 *El estado de un objeto de la clase* `coche` *podría ser:*

```
{<matricula,A-6065-CK>, <velocidad,80>,
<temperatura_del_motor,normal>}
```

El estado permite caracterizar a un objeto en un instante determinado. Llamaremos Σ al conjunto de estados alcanzables por un objeto y $\sigma \in \Sigma$ a un estado en particular. El estado de un objeto cambia por la ocurrencia de acciones[3]. Los cambios de estado son modelados por la siguiente relación:

$$efecto : \underline{A} \rightarrow (\Sigma \rightarrow \Sigma)$$

Donde $\underline{A}$ es el conjunto de acciones instanciadas ($\underline{a} \in \underline{A}$) que le pueden acontecer al objeto y $efecto$ es el cambio de estado atómico que produce la ocurrencia de una acción. En un mismo instante de tiempo, a un objeto le puede acontecer más de una acción.

[3]Utilizaremos la *frame assumption*, es decir, supondremos que si no se especifica que la modificación del valor de un atributo se debe a la ocurrencia de una determinada acción, entonces dicho atributo no es modificado por dicha acción.

Definición 7 ***Conjunto consistente de acciones****. Un conjunto de acciones $M \subseteq \underline{A}$ es consistente si a partir de un estado σ dado, el estado σ' alcanzado por la ocurrencia de las acciones en M es siempre el mismo, independiente del orden en el cual se ejecuten*[4] *las acciones.*

Definición 8 ***Paso****. Llamaremos paso a un conjunto consistente de acciones que ocurren en un mismo instante de la vida de un objeto.*

Sea $\mu = \{a_1, ..., a_n\} \subseteq \underline{A}$ un paso. La relación de accesibilidad entre estados (R_μ) está determinada por μ, $R_\mu \subseteq \Sigma \times \Sigma$ y se define como:

$$R_\mu(\sigma, \sigma') \stackrel{def}{\equiv} efecto(a_1)(\sigma) \circ ... \circ efecto(a_n)(\sigma) = \sigma'$$

De acuerdo con la definición de conjunto consistente, el orden en el cual se aplica la función $efecto$ no altera el estado siguiente alcanzado por el objeto.

Definición 9 ***Vida o traza de un objeto****. Llamaremos vida o traza de un objeto a un prefijo finito de pasos que le acontecen a dicho objeto.*

En OASIS, el modelado del comportamiento puede realizarse desde dos perspectivas complementarias en cuanto a expresividad. En cada instante, las acciones permitidas y las acciones obligatorias para un objeto pueden establecerse de la siguiente forma:

- Considerando el estado del objeto en dicho instante. Las precondiciones son fórmulas que permiten la ocurrencia de acciones sólo en ciertos estados del objeto. De forma análoga, los disparos son fórmulas que obligan a la ocurrencia de acciones en determinados estados del objeto.

- Considerando las acciones ya acontecidas. Una especificación de proceso determina un patrón de comportamiento para el objeto desde el punto de vista de secuencias de pasos acontecidos, es decir, determina trazas posibles. En OASIS se utilizan dos tipos de especificaciones de proceso: *protocolos* y *operaciones*. Un protocolo establece secuencias permitidas de acciones. Una operación establece secuencias obligatorias de acciones.

[4] Utilizaremos "ejecución de acción" como sinónimo de "ocurrencia de acción" para hacer más intuitiva la presentación. Sin embargo, desde el punto de vista declarativo y formal, "ocurrencia" es el término más adecuado.

2.4. OASIS expresado en Lógica Dinámica

De forma global, puede verse que OASIS está basado en Lógica Deóntica [2]. La Lógica Deóntica es la lógica de obligaciones, prohibiciones y permisos. Siendo A el conjunto de acciones con $a \in A$, en OASIS se utilizan los siguientes operadores de la Lógica Deóntica:

$O(a)$	obligación de ocurrencia de a.
$F(a)$	prohibición de ocurrencia de a.
$P(a)$	permiso o habilitación de ocurrencia de a.

En [20] la Lógica Deóntica es descrita como una variante de Lógica Dinámica [13]. La Lógica Dinámica es un formalismo natural para estudiar de forma simple y directa ciertas aserciones sobre programas. En Lógica Dinámica, la fórmula $[a]\phi$ se interpreta intuitivamente como "después de la ocurrencia de la acción a, ϕ debe satisfacerse", siendo a una acción y ϕ una Fbf de la Lógica de Primer Orden utilizada. La definición de los operadores deónticos en Lógica Dinámica es la siguiente:

$O(a) \Leftrightarrow [\neg a]\ false$	"la ocurrencia de a es obligatoria".
$F(a) \Leftrightarrow [a]\ false$	"la ocurrencia de a está prohibida".
$P(a) \Leftrightarrow \neg F(a)$	"a está permitida si y sólo si a no está prohibida".

Donde $\neg a$ representa la no-ocurrencia de la acción a (es decir, que no ocurre nada o que ocurre otra acción distinta de a). En estas fórmulas, $false$ es un átomo tal que no hay un estado que pueda hacerlo verdadero. El significado intuitivo de éstas fórmulas es: una acción está prohibida si su ocurrencia conduce a un estado de violación. Una acción es obligatoria si su no-ocurrencia conduce a un estado de violación.

Siendo ψ una Fbf que caracteriza el estado del objeto, utilizaremos los siguientes tipos de fórmulas en Lógica Dinámica:

$\psi \rightarrow [a]false$	"la ocurrencia de a está prohibida en estados que satisfacen ψ".
$\psi \rightarrow [\neg a]false$	"la ocurrencia de a es obligatoria en estados que satisfacen ψ".
$\psi \rightarrow [a]\phi$	"en estados que satisfacen ψ, justo después de la ocurrencia de la acción a, ϕ debe satisfacerse".

Estas fórmulas constituyen un sublenguaje del lenguaje de Lógica Dinámica propuesto y formalizado en [33].

A continuación se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase en OASIS. Posteriormente se describe cómo la plantilla completa se corresponde con fórmulas en la variante de Lógica Dinámica que se ha esbozado. Como veremos, la correspondencia es directa, salvo para especificaciones de proceso.

2.5. Fórmulas de especificación

Para describir las fórmulas usadas en cada sección de una plantilla de clase, consideraremos ϕ, ϕ' y ψ como Fbf en la Lógica de Predicados de Primer Orden usada, con $a \in A$.

Evaluaciones

Estas fórmulas definen los cambios de estado ante la ocurrencia de acciones. Una evaluación se corresponde con la siguiente fórmula de la Lógica Dinámica utilizada:

$$\psi \rightarrow [a]\phi$$

En este caso, ϕ es una Fbf construida usando sólo átomos con el operador relacional '=' y conectivas *and*. Con esta sintaxis es posible caracterizar explícitamente parte del estado de un objeto antes y después de la ocurrencia de una determinada acción.

Derivaciones

Las fórmulas usadas en derivaciones (información derivada) tienen la forma: $\phi \rightarrow \phi'$. Una fórmula de derivación se debe satisfacer en cada estado del objeto. ϕ' es una fórmula que expresa mediante una igualdad cómo se obtiene el valor del atributo derivado. En Lógica Dinámica, las fórmulas para derivaciones pueden ser representadas como:

$$[a](\phi \rightarrow \phi'), \forall a \in A.$$

Precondiciones

Son fórmulas en la variante de Lógica Dinámica utilizada. Representan prohibiciones para la ocurrencia de acciones. Así, una precondición es definida por la fórmula:

$\psi \rightarrow [a]false$ "si ψ se satisface entonces la ocurrencia de a está prohibida".

Como se mencionó antes, los permisos (o habilitación de ocurrencia de acciones) constituyen una subespecificación, es decir, si una acción no está prohibida, entonces está permitida.

Por comodidad en la especificación, ψ será una fórmula del tipo $\neg\phi$, pues resulta más natural expresar "si $\neg\phi$ se satisface entonces a es prohibido". Así, la fórmula en Lógica Dinámica para precondiciones se escribe finalmente como:

$\neg\phi \rightarrow [a]false$ "si $\neg\phi$ se satisface entonces la ocurrencia de a está prohibida".

Disparos

Los disparos permiten especificar actividad basada en el estado del objeto. Un disparo es una obligación y se define con la fórmula:

$\psi \rightarrow [\neg a]false$ "si ψ se satisface entonces la ocurrencia de a es obligatoria".

Restricciones de integridad

Son Fbfs en Lógica Temporal usadas para representar restricciones de integridad. Obedecen a las siguientes reglas:

- Toda Fbf ϕ es una Fbf en Lógica Temporal interpretada como *always* ϕ. Además, *always* ϕ, *sometimes* ϕ, *next* ϕ, ϕ *since* ϕ' y ϕ *until* ϕ' son también Fbf en Lógica Temporal.

- Sea T un periodo de tiempo expresado usando las unidades de tiempo usuales: *seconds*, *minutes*, *hours*, *days*, *weeks*, *months* o *years*, entonces:

$sometimes_{op\_rel\ T}\ \ \phi\ since\ \phi'$

$always_{op\_rel\ T}\ \ \phi\ since\ \phi'$

Ambas son Fbf en Lógica Temporal, siendo op_rel el operador relacional $<$ ó $\leqslant$ en la primera fórmula (para expresar un *timeout*) y los operadores $>$ ó $\geqslant$ en la última (para expresar un *delay*).

- Sólo son Fbf en Lógica Temporal válidas para esta sección las construidas siguiendo las reglas anteriores.

Si ϕ es una Fbf en la Lógica Temporal presentada, entonces desde la Lógica Dinámica, dicha fórmula es vista como un conjunto de fórmulas del tipo : $[a]\phi, (a \in A)$

2.6. Especificación de procesos

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 [21], 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 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

del "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.

Siguiendo el planteamiento y notación propuestos en CCS, la especificación de un proceso R puede ser vista como un sistema de transiciones etiquetado representado por $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$, donde:

- κ_R es el conjunto de constantes agente[5] utilizados para definir el proceso R.
- A_R es el conjunto de acciones usadas en la especificación del proceso R, con $A_R \subseteq A$, y siendo A el conjunto de acciones[6] en la signatura de la clase.
- $\xrightarrow{a} \subseteq \kappa_R \times \kappa_R$ es la relación de transición.

Para cada constante agente $E \in \kappa_R$ se tiene una ecuación de definición $E \stackrel{def}{\equiv} Q$, siendo Q una expresión construida usando los siguientes operadores (con $S \in \kappa_R$ y a, a_1 y a_2 acciones pertenecientes al conjunto A_R):

1. $a.S$, es un prefijo.
2. $if\ C\ then\ a.S$, es un condicional, siendo C una expresión *boolean*.

   Extensiones del tipo $if\ C\ then\ a_1.S_1\ else\ a_2.S_2$ pueden ser definidas reescribiéndolas como $(if\ C\ then\ a_1.S_1) + (if\ \neg C\ then\ a_2.S_2)$.

3. $\sum_{i \in I} Q'_i$, donde Q'_i puede obtenerse por alguno de los operadores anteriores y $\sum_{i \in I}$ es la suma con I como conjunto de indexación.

Existe una constante adicional e implícita denotada por $\mathbf{0}$ y definida como $\mathbf{0} = \sum_{i \in I} a_i.S_i$, con $I = \emptyset$. Por lo tanto, desde $\mathbf{0}$ no existen transiciones posibles.

[5] Un proceso de OASIS corresponde al concepto de agente de CCS. Un agente está representado por una constante agente (su nombre). En la definición de una constante agente se pueden utilizar otras constantes agente.

[6] Incluye tanto las acciones generadas desde otros objetos como las acciones generadas por el propio objeto. Es decir, el objeto visto como cliente y servidor.

Ejemplo 3 *Sea un proceso P especificado como:*

$$\begin{array}{lcl} P & \stackrel{def}{\equiv} & a.P_1 \\ P_1 & \stackrel{def}{\equiv} & b.P_2 + e.P_3 \\ P_2 & \stackrel{def}{\equiv} & c.P_2 + b.P_2 + d.P_1 \\ P_3 & \stackrel{def}{\equiv} & \mathbf{0} \end{array}$$

El sistema de transiciones para el proceso P es:

$$\begin{array}{lcl} \kappa_P & = & \{P, P_1, P_2, P_3\} \\ \\ A_P & = & \{a, b, c, d, e\} \\ \\ \stackrel{a}{\rightarrow} & = & \{(P, P_1)\} \\ \stackrel{b}{\rightarrow} & = & \{(P_1, P_2), (P_2, P_2)\} \\ \stackrel{c}{\rightarrow} & = & \{(P_2, P_2)\} \\ \stackrel{d}{\rightarrow} & = & \{(P_2, P_1)\} \\ \stackrel{e}{\rightarrow} & = & \{(P_1, P_3)\} \end{array}$$

Un proceso puede ser representado por un grafo de transiciones. La Figura 2.1 muestra el grafo de transiciones para el proceso P del ejemplo.

Definición 10 ***Estado del proceso****. Llamamos estado del proceso a la constante agente que en un determinado instante representa al proceso y que se corresponde con un nodo del grafo de transiciones del proceso.*

En la plantilla de clase OASIS, para cada proceso R existe implícitamente un atributo variable de *sort* κ_R con el mismo nombre del proceso. Dicho atributo representa el estado del proceso (la constante agente o nodo del grafo de transiciones asociado) en el cual se encuentra el proceso. Como veremos, cuando el objeto está en el estado **0** en un determinado proceso, el comportamiento del objeto no está siendo afectado por el proceso (el proceso no determina ni permisos ni obligaciones para el objeto).

Es importante destacar que un mismo estado del proceso puede estar asociado a más de un estado del objeto (un σ distinto).

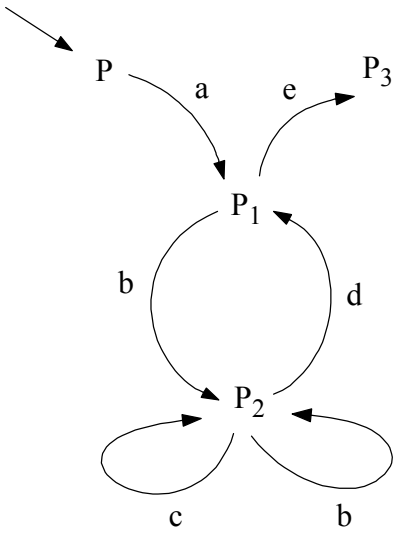

Figura 2.1: Grafo de transiciones del proceso P.

2.6.1. Procesos como fórmulas en Lógica Dinámica

Como se ha expuesto, los operadores de la Lógica Deóntica que expresan obligaciones, prohibiciones y permisos pueden ser vistos como una variante de Lógica Dinámica. Esto sugiere que los procesos de obligación y prohibición se corresponden del mismo modo con fórmulas en Lógica Dinámica.

Una especificación de proceso determina una conducta particular del objeto. Diremos que *un proceso afecta a un objeto* si el objeto debe comportarse según la conducta establecida por dicho proceso. Comúnmente, el objeto seguirá la conducta del proceso en partes de su vida. Un caso particular es un proceso que afecta al objeto durante toda su existencia.

Un proceso que establece permisos se puede expresar como un conjunto de fórmulas en Lógica Dinámica de la forma $\neg\phi \rightarrow [a]false$, del mismo modo que se indicó para las fórmulas de precondiciones.

De forma análoga, un proceso que se establece por obligaciones se puede expresar como un conjunto de fórmulas en Lógica Dinámica de la forma $\psi \rightarrow [\neg a]false$, tal como se indicó para las fórmulas de disparos.

Como veremos, en ambos casos se obtendrán adicionalmente fórmulas por transición de estado del proceso. Estas fórmulas son del mismo tipo

usado para evaluaciones, es decir, $\psi \rightarrow [a]\phi$.

Sea R un proceso definido por el sistema de transiciones $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$, E,$E' \in \kappa_R$. Si $(E, E') \in \xrightarrow{a}$, esto suele escribirse como $E \xrightarrow{a} E'$ y se dice que a **es una acción desde** E.

Ejemplo 4 *Utilizaremos el proceso P presentado en el ejemplo 3. La relación de transición de P es:*

$$\begin{array}{rcl} \xrightarrow{a} & = & \{(P, P_1)\} \\ \xrightarrow{b} & = & \{(P_1, P_2), (P_2, P_2)\} \\ \xrightarrow{c} & = & \{(P_2, P_2)\} \\ \xrightarrow{d} & = & \{(P_2, P_1)\} \\ \xrightarrow{e} & = & \{(P_1, P_3)\} \end{array}$$

Por lo tanto, en P:

a es una acción desde P.
b es una acción desde P_1 y desde P_2.
c es una acción desde P_2.
d es una acción desde P_2.
e es una acción desde P_1.

Definición 11 ***Predicado de ocurrencia de una acción en un estado del proceso****. Sea r el atributo variable asociado al estado del proceso R. Si a es una acción desde un estado E del proceso, llamaremos* ***predicado de ocurrencia de*** *a* ***en*** *E al predicado $((r = E) \wedge C)$ y lo denotaremos por Ω_{Ea}, donde C es una expresión boolean si para (E, E') está definida. De lo contrario, el predicado de ocurrencia de a en E es sólo $(r = E)$. Por defecto, si a no es una acción desde E, entonces $\Omega_{Ea} \equiv false$.*

Ejemplo 5 *Los predicados de ocurrencia de cada acción en cada estado del proceso P son:*

$$\begin{array}{rcl} \Omega_{Pa} & \equiv & (p = P) \\ \Omega_{P_1 b} & \equiv & (p = P_1) \\ \Omega_{P_2 b} & \equiv & (p = P_2) \\ \Omega_{P_2 c} & \equiv & (p = P_2) \\ \Omega_{P_2 d} & \equiv & (p = P_2) \\ \Omega_{P_1 e} & \equiv & (p = P_1) \end{array}$$

Para el resto de combinaciones entre estado del proceso y acciones, el predicado de ocurrencia es $false$.

Definición 12 ***Predicado de ocurrencia de una acción en el proceso.*** *Ampliando el concepto anterior al proceso como un todo, llamaremos* ***predicado de ocurrencia de*** *a* ***en*** *R, denotado por Ω_{R^*a}, a la disyunción de predicados de ocurrencia de a desde E, para cada constante agente E que define al proceso R.*

Ejemplo 6 *Los predicados de ocurrencia en P para cada una de sus acciones, son:*

$$\begin{array}{lcl} \Omega_{P^*a} & \equiv & (p = P) \\ \Omega_{P^*b} & \equiv & ((p = P_1) \vee (p = P_2)) \\ \Omega_{P^*c} & \equiv & (p = P_2) \\ \Omega_{P^*d} & \equiv & (p = P_2) \\ \Omega_{P^*e} & \equiv & (p = P_1) \end{array}$$

El significado de Ω_{R^*a} es el siguiente: "Ω_{R^*a} se satisface si y sólo si el proceso R está en algún estado de proceso E y algún Ω_{Ea} se satisface, es decir, a es una acción desde E y además se cumple la condición C asociada (si ésta existe)".

Así, un proceso R definido por el sistema de transiciones $(\kappa_R, A_R, \{\stackrel{a}{\rightarrow}: a \in A_R\})$, se corresponde con las fórmulas en Lógica Dinámica presentadas a continuación.

Fórmulas por transiciones. Efectúan la evolución del proceso de acuerdo con las transiciones definidas y con la ocurrencia de acciones. Para cada $a \in A_R$, y para cada $(E, E') \in \stackrel{a}{\rightarrow}$ se obtiene la siguiente fórmula:

$$\Omega_{Ea} \rightarrow [a](r = E')$$

Ejemplo 7 *Las fórmulas por transiciones para el proceso P son:*

$$\begin{array}{lcl} (p = P) & \rightarrow & [a](p = P_1) \\ (p = P_1) & \rightarrow & [b](p = P_2) \\ (p = P_2) & \rightarrow & [b](p = P_2) \\ (p = P_2) & \rightarrow & [c](p = P_2) \\ (p = P_2) & \rightarrow & [d](p = P_1) \\ (p = P_1) & \rightarrow & [e](p = P_3) \end{array}$$

El grafo de alcanzabilidad que determinan estas transiciones corresponde al grafo de transiciones mostrado en la Figura 2.1.

Además, siendo a_{new} la acción de creación para instancias de la clase, entonces:

a) Si R es un proceso que establece obligaciones entonces, adicionalmente, existirá la fórmula:

$[a_{new}](r = \mathbf{0})$
Es decir, asumimos que un proceso de obligación no afecta al objeto desde el comienzo de su existencia.

b) Si R es un proceso que establece permisos, la fórmula adicional es:

$[a_{new}](r = E_1)$

$E_1 \in \kappa_R$ es la constante agente de inicio del proceso.

Es decir, asumimos que un proceso de permiso afecta al objeto desde el comienzo de su existencia.

Fórmulas por Permisos (sólo si R es un proceso que establece permisos). De acuerdo con el estado del proceso, una fórmula de permiso establece la situación en la cual la ocurrencia de una acción se permite. Para cada $a \in A_R$ se obtiene la siguiente fórmula:

$$\neg\Omega_{R^*a} \rightarrow [a]false.$$

Ejemplo 8 *Si en el ejemplo tratado el proceso P fuera un proceso que establece secuencias de acciones permitidas, las fórmulas de permiso asociadas serían:*

$$\begin{array}{lcl} \neg(p = P) & \rightarrow & [a]false \\ \neg((p = P_1) \vee (p = P_2)) & \rightarrow & [b]false \\ \neg(p = P_2) & \rightarrow & [c]false \\ \neg(p = P_2) & \rightarrow & [d]false \\ \neg(p = P_1) & \rightarrow & [e]false \end{array}$$

Fórmulas por Obligaciones (sólo si R es un proceso que establece obligaciones). De acuerdo con el estado del proceso, una fórmula de obligación establece la situación en la cual la ocurrencia de una acción es obligatoria. Para cada $a \in A_R$ se obtiene la siguiente fórmula:

$$\Omega_{R^*a} \rightarrow [\neg a] false.$$

Ejemplo 9 *Si en el ejemplo tratado, el proceso P fuera un proceso que establece secuencias de acciones obligatorias, las fórmulas de obligación asociadas serían:*

$$\begin{array}{lcl} (p = P) & \rightarrow & [\neg a] false \\ ((p = P_1) \vee (p = P_2)) & \rightarrow & [\neg b] false \\ (p = P_2) & \rightarrow & [\neg c] false \\ (p = P_2) & \rightarrow & [\neg d] false \\ (p = P_1) & \rightarrow & [\neg e] false \end{array}$$

2.6.2. Aspectos adicionales

Manteniendo el marco formal establecido, en este apartado se presentan aspectos adicionales que extienden la capacidad expresiva que puede ser usada en especificaciones de proceso.

Tratamiento de pasos y operaciones

Un paso es un conjunto consistente de acciones que acontecen en un instante dado en la vida de un objeto. Una acción está asociada a un servicio. El servicio puede ser un evento o una operación. El evento no tiene duración, es instantáneo. En cambio, una operación establece una secuencia obligatoria de acciones y, en general, tiene una duración.

El cliente no tiene porqué conocer si un servicio corresponde a un evento o una operación en el servidor. Por esto, en la acción acontecida en el cliente, y que refleja la solicitud del servicio, no se hace referencia alguna que distinga entre solicitar un evento o solicitar una operación.

En el servidor, el tratamiento para servir una operación es distinto al dado cuando el servicio es un evento. Al proveer operaciones consideraremos lo siguiente:

- Se asume que las precondiciones o evaluaciones (si existen) asociadas a una operación son precondiciones o evaluaciones para el evento de inicio de la operación.

- Para asegurar la correcta realización de una operación, mientras una operación esté en ejecución, no deben servirse otros eventos ni operaciones que hagan incompatible el concepto de paso (conjunto consistente de acciones) con la obligatoriedad de ejecución de las acciones de la operación[7].

Una operación puede ser vista como una secuencia de acciones (las que determina su especificación) más dos acciones implícitas: la acción de inicio de la operación y la acción de término de la operación. La acción de inicio tiene siempre un evento requerido por un determinado cliente (quien solicita la operación). La acción de término tiene siempre un evento requerido a sí mismo, obligado e inmediatamente posterior a la ejecución de la última acción ejecutada por la operación.

El mantener implícito las acciones de inicio y término de operación simplifica la especificación. Al mismo tiempo, apoya la idea de que tanto eventos como operaciones sean tratados en la especificación bajo un concepto más amplio; el concepto de servicio. A continuación, se abordará el tratamiento de operaciones, sólo para mostrar la correspondencia con la formalización de procesos ya realizada (en la que no se discutió el tratamiento de operaciones).

Sea op una operación definida para el objeto. Denotaremos por $op^{\bullet}$ a la acción de inicio de la operación y por $op^{\odot}$ a la acción de término de la operación.

Si existiera una precondición como: $\neg\phi \rightarrow [op]false$, se interpretaría implícitamente como una precondición para la acción de inicio de la operación, es decir, $\neg\phi \rightarrow [op^{\bullet}]false$. Del mismo modo, una evaluación del tipo $\psi \rightarrow [op]\phi$ se interpreta como una evaluación asociada a la acción de inicio de la operación, es decir, $\psi \rightarrow [op^{\bullet}]\phi$.

Según lo anterior, el tratamiento de operaciones se reduce a la ejecución de las acciones de la operación más las dos acciones implícitas de inicio y fin de la operación.

Queda por asegurar que al permitirse concurrencia intra-objetual (de eventos y operaciones), las acciones de un paso sean siempre consistentes y a la vez se cumpla la obligatoriedad de las acciones de una operación en ejecución.

[7]El problema es análogo al de control de concurrencia de transacciones, ampliamente tratado en bases de datos relacionales. Nótese que la definición previa de paso era aplicable sólo para acciones ejecutadas en un mismo instante. Este concepto debe ser extendido para considerar cierta duración asociada a la ejecución de una operación.

La no-consistencia entre acciones es difícil de establecer. Sin embargo, la consistencia se cumple siempre si las acciones no están en conflicto[8] [17]. El conjunto de atributos sobre el cual puede tener efecto una operación está determinado por la unión de los conjuntos de atributos sobre los cuales tienen efecto las acciones incluidas en la operación.

Asociado a cada operación, existe un atributo implícito del objeto que determina en qué estado de la operación se encuentra el objeto. Si la operación no está en ejecución entonces dicho atributo toma el valor **0**. Siendo *op* el atributo que registra el estado del proceso *op*, entonces existe una evaluación implícita del tipo $[op^{\odot}](op = \mathbf{0})$. Esta evaluación hace que inmediatamente después de ejecutarse la acción $op^{\odot}$ (acción de término del proceso), el proceso *op* se encuentre en su estado de proceso **0**, es decir, el proceso vuelve a una situación de inactividad.

Por lo tanto, para asegurar que los eventos y operaciones (tratadas como acciones) de un paso no estén en conflicto, además de verificar que no estén en conflicto entre ellas, debe verificarse que no estén en conflicto con ninguna operación en ejecución (cuyo valor de atributo asociado es diferente a **0**).

Capacidad para especificar procesos anidados

Cuando las especificaciones de procesos son extensas (muchos estados de proceso) es conveniente disponer de algún mecanismo de descomposición. Así, un proceso puede ser descompuesto en subprocesos anidados[9] especificados separadamente. Esto a su vez promueve la reutilización de especificaciones de subprocesos desde distintos procesos definidos en la plantilla de una clase.

Ejemplo 10 *Consideremos la especificación de un proceso L. Por simplicidad, supondremos que las acciones a_1 y a_2 son acciones cuyo servicio es un evento.*

[8] Dos acciones no están en conflicto si tienen efecto sobre conjuntos disjuntos de atributos. Este criterio es bastante restringido respecto de la concurrencia intra-objetual. Sin embargo, es fácil de determinar: basta inspeccionar evaluaciones asociadas a las acciones y posibles derivaciones de atributos.

[9] Por *anidamiento* entendemos la capacidad expresiva que permite que un proceso en su definición haga referencia a otro proceso como una forma de transición compleja, definida por otro proceso.

$$
\begin{array}{lcl}
L & \stackrel{def}{\equiv} & a_1.L_1 \\
L_1 & \stackrel{def}{\equiv} & a_2.L_1 + Q^{\bullet}.Q \\
L_2 & \stackrel{def}{\equiv} & a_1.L_3 \\
L_3 & \stackrel{def}{\equiv} & \mathbf{0} \\
Q & \stackrel{def}{\equiv} & a_2.Q_1 + a_1.Q_2 \\
Q_1 & \stackrel{def}{\equiv} & a_1.Q + a_2.Q_2 \\
Q_2 & \stackrel{def}{\equiv} & Q^{\odot}.L_2
\end{array}
$$

$Q^{\bullet}$ y $Q^{\odot}$ son dos acciones. Dichas acciones pueden verse como las acciones de inicio y de término para un cierto proceso (Q), respectivamente.

Esto sugiere reescribir L como:

$$
\begin{array}{lcl}
L & \stackrel{def}{\equiv} & a_1.L_1 \\
L_1 & \stackrel{def}{\equiv} & a_2.L_1 + q.L_2 \\
L_2 & \stackrel{def}{\equiv} & a_1.L_3 \\
L_3 & \stackrel{def}{\equiv} & \mathbf{0}
\end{array}
$$

Donde q es una acción que incluye al servicio no atómico Q. Así, el proceso L es definido utilizando la especificación del proceso Q, reescrito como:

$$
\begin{array}{lcl}
Q & \stackrel{def}{\equiv} & a_2.Q_1 + a_1.Q_2 \\
Q_1 & \stackrel{def}{\equiv} & a_1.Q + a_2.Q_2 \\
Q_2 & \stackrel{def}{\equiv} & \mathbf{0}
\end{array}
$$

El proceso Q podría entonces ser utilizado en otras definiciones de proceso. A su vez, el proceso Q también podría utilizar otras definiciones de proceso.

Así, la justificación de este anidamiento de especificaciones de proceso se obtiene estableciendo la correspondencia de la especificación anidada con una equivalente sin anidamiento.

2.7. Plantilla de clase en Lógica Dinámica

Después de lo presentado, expresar la plantilla de una clase OASIS como un conjunto de fórmulas en Lógica Dinámica es inmediato. Siendo $\langle Atributos, Eventos, Fórmulas, Procesos\rangle$ la plantilla de una clase, el conjunto de fórmulas en Lógica Dinámica para una clase es el siguiente:

i. Para cada evaluación existirá una fórmula de cambio de estado:

$\psi \rightarrow [a]\phi$

ii. Para cada $a \in A$ y para cada fórmula de derivación $(\phi \rightarrow \phi')$ existirá una fórmula del tipo:

$[a](\phi \rightarrow \phi')$

iii. Para cada precondición existirá una fórmula de permiso:

$\neg\phi \rightarrow [a]false$

iv. Para cada disparo existirá una fórmula de obligación:

$\phi \rightarrow [\neg a]false$

v. Para cada[10] $a \in A$ y siendo φ la conjunción de todas las fórmulas de restricciones de integridad existirá una fórmula del tipo:

$[a]\varphi$

vi. Para cada proceso de obligación R, definido por $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$

- Fórmulas por transiciones

Para cada $(E, E') \in \xrightarrow{a}$ la fórmula: $\Omega_{Ea} \rightarrow [a](r = E')$.

Además, la fórmula: $[a_{new}](r = \mathbf{0})$.

[10] Que ocurre fuera del contexto de una transacción o que representa a la acción de fin de proceso de la transacción.

- Fórmulas por obligaciones

  Para cada $a \in A_R$ la fórmula: $\Omega_{R^*a} \rightarrow [\neg a] false$.

vii. Para cada proceso de permiso R, definido por $(\kappa_R, A_R, \{\xrightarrow{a}: a \in A_R\})$

- Fórmulas por transiciones

  Para cada $(E, E') \in \xrightarrow{a}$ la fórmula: $\Omega_{Ea} \rightarrow [a](r = E')$.

  Además, la fórmula: $[a_{new}](r = E_1)$, $E_1 \in \kappa_R$ es el estado de inicio del proceso.

- Fórmulas por permisos

  Para cada $a \in A_R$, la fórmula: $\neg\Omega_{R^*a} \rightarrow [a] false$.

2.8. 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[11] posibles que un objeto puede alcanzar. Sea F el conjunto de Fbf evaluadas sobre el estado (mundo asociado) en el cual se encuentra el objeto, A el conjunto de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos 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 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 $\mu \in 2^{\underline{A}}$ 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'.

Definición 13 ***Transición válida.*** *Cada par $(w, w') \in \rho(\mu)$ se denomina llama transición válida, con $w, w' \in W$ y $\mu \in 2^{\underline{A}}$.*

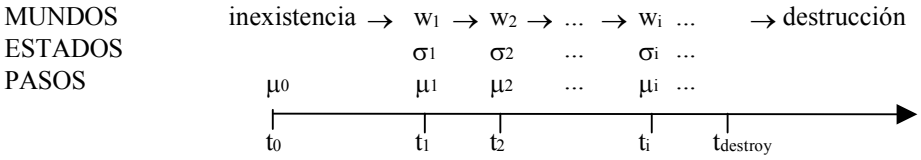

Figura 2.2: Relación entre mundos, estados y pasos del objeto.

La relación entre estados, mundos, pasos y la vida del objeto se representa en la Figura 2.2 donde t_i es el tiempo en el cual se produce el i-ésimo *tic* de reloj para el objeto.

Obviamente, se cumple que: $\models_{w_i} \sigma_i$, $i = 1, ..., n$. 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 permiso $\neg\phi \rightarrow [a]false$

   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 $\models_w \neg\phi$.

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

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

   Siendo $w \in \tau(\neg\phi)$ 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 $\models_w \phi$.

Capítulo 3

Clases Complejas

Una clase compleja es una clase que utiliza otras clases (elementales o complejas) en su definición. La complejidad es inherente al modelado de sistemas de información, así, un lenguaje de especificación OO debe proporcionar constructores de clases complejas que combinen la expresividad adecuada con la facilidad de uso. En OASIS estos constructores se denominan operadores de clase.

Los mecanismos utilizados para formar clases complejas están relacionados con las nociones de agregación y herencia. La agregación expresa cómo diversos objetos pueden agruparse formando nuevos objetos y coordinando sus vidas. Mediante herencia (especialización y rol) se expresa cómo pueden definirse nuevas clases heredando propiedades (estructura y comportamiento) de otras clases.

3.1. Morfismos entre objetos

La semántica asociada a objetos de clases complejas puede ser abordada como una combinación de procesos desde la Teoría de Procesos. Los **morfismos** entre objetos permiten que sean combinados mediante relaciones que preservan la estructura y el comportamiento. Preservar el comportamiento significa que existe un morfismo entre aquellos comportamientos que estén relacionados y que cumple ciertas propiedades. Dichos morfismos van a permitir definir la herencia y relaciones de agregación (entre objetos componentes para formar objetos agregados).

Parte de las definiciones presentadas en este capítulo han sido recogidas desde [9] y adaptadas para nuestro modelo. Hemos visto que el com-

portamiento de un objeto viene determinado por el conjunto de trazas que le pueden acontecer. También sabemos que el alfabeto de las trazas está definido sobre el conjunto de acciones en la signatura de la clase del objeto.

Además de los servicios requeridos u ofrecidos que representan las acciones de un objeto, debemos considerar la existencia de *observaciones* que se pueden hacer de un objeto. Estas observaciones no tienen efecto sobre los valores de los atributos.

Un objeto puede ser visto como una relación entre las observaciones y las acciones que le pueden acontencer a lo largo del tiempo. Estas observaciones dependen del estado del objeto, en particular podrían ser los atributos evaluados del objeto. Podemos revisitar el concepto de objeto definiéndolo como una relación entre acciones y observaciones, o dicho de otro modo, como un *proceso observable* [26].

Definición 14 ***Objeto.*** *Un objeto es un morfismo de comportamiento* $ob : (A, \Lambda) \rightarrow (V, \vartheta)$*, donde* A *es el alfabeto de pasos incluyendo sólo acciones,* Λ *son las trazas posibles,* V *es el alfabeto de pasos incluyendo sólo observaciones y* ϑ *son las observaciones posibles. Escribiremos* $ob : \Lambda \rightarrow \vartheta$ *si el contexto está sobreentendido.*

Existen dos formas básicas de derivar nuevos objetos a partir de otros existentes: (1) por observación y (2) por trayectoria, a través de morfismos de estado (h_ϑ) y de comportamiento (h_Λ), tal como se indica en la Figura 3.1.

Sean $objeto\_A : \Lambda_1 \rightarrow \vartheta_1$, $objeto\_B : \Lambda_2 \rightarrow \vartheta_2$ dos objetos cualesquiera. En la Figura 3.1(a) $objeto\_A$ ha sido derivado por trayectoria a través de h_Λ con lo que $objeto\_A \circ h_\Lambda$ tiene Λ_2 como parte de proceso y h_Λ indica a $objeto\_A$ cómo debe interpretar las acciones de Λ_2.

De forma análoga, en la Figura 3.1(b) $objeto\_B$ es observado vía h_ϑ. Es decir, h_ϑ dice cómo interpretar las observaciones de $objeto\_B$ en términos de la conducta de $objeto\_A$. Para el caso particular en el que h_ϑ y h_Λ son restricciones en los alfabetos de pasos respectivos (es decir, inclusiones en los alfabetos básicos), la derivación por trayectoria significa suprimir de Λ_2 las acciones que no están en Λ_1 e interpretar las observaciones significa ver sólo aquello que está en el ámbito de ϑ_1.

Un morfismo objetual es una relación entre los objetos $objeto\_A$ y $objeto\_B$ donde la parte traza de $objeto\_B$ deriva la trayectoria de

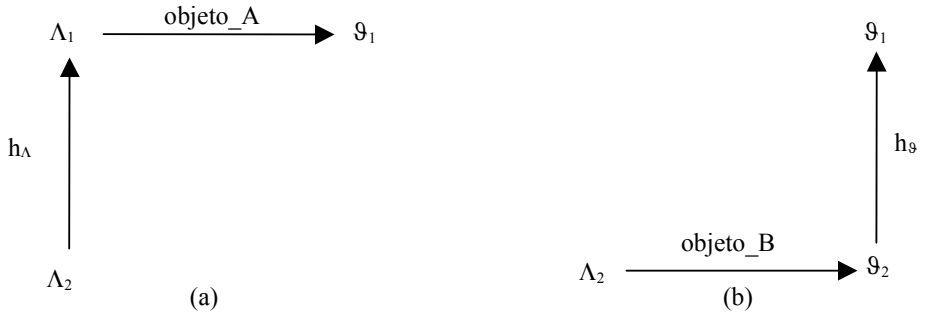

Figura 3.1: Formas de derivación de objetos.

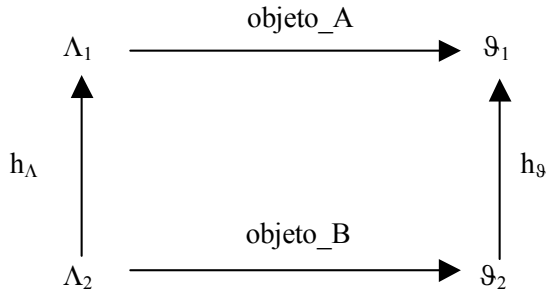

Figura 3.2: Diagrama de morfismo entre objetos.

objeto_A a través de algún morfismo de comportamiento h_Λ y, al mismo tiempo, la parte de observación de *objeto_A* se obtiene de las observaciones de *objeto_B* a través de algún otro morfismo h_ϑ, de tal manera que *objeto_A* derivado a través de h_Λ es el mismo objeto que *objeto_B* observado a través de h_ϑ, es decir, que el diagrama de la Figura 3.2 conmuta.

Definición 15 ***Morfismo entre objetos.*** *Siendo objeto_A* $: \Lambda_1 \rightarrow \vartheta_1$, *objeto_B* $: \Lambda_2 \rightarrow \vartheta_2$ *dos objetos, entonces, un morfismo entre objetos h* : *objeto_B* $\rightarrow$ *objecto_A es un par de morfismos, uno de comportamiento* $h_\Lambda : \Lambda_2 \rightarrow \Lambda_1$ *y otro de estado* $h_\vartheta : \vartheta_2 \rightarrow \vartheta_1$, *tales que se cumple:*

$$objeto\_A \circ h_\Lambda = h_\vartheta \circ objeto\_B$$

Los objetos individuales se describen como morfismos, la interacción entre objetos se describe mediante morfismos entre objetos y la obtención de objetos compuestos mediante comunidades de objetos que interactúan entre sí. A continuación veremos distintos tipos de composición de objetos.

3.2. Inclusión fuerte

Intuitivamente, un morfismo de inclusión fuerte[1] h : *componente* $\hookrightarrow$ *agregado* describe cómo la parte componente queda incluida en un objeto agregado tal que dicho componente está encapsulado dentro del agregado de manera que ninguna acción exterior puede afectarle de forma directa. Ejemplos típicos de encapsulación son los siguientes:

`motor`$\hookrightarrow$`coche`, `memoria`$\hookrightarrow$`computador`, ...

Un caso particular de este tipo de morfismo entre objetos es la compartición de objetos, es decir, la inclusión de un objeto dentro de varios. En esta situación, los servicios son compartidos de forma síncrona y simétrica entre los objetos.

Definición 16 ***Morfismo de inclusión fuerte.*** *Siendo $h : ob_1 \rightarrow ob_2$ un morfismo entre objetos. Si el morfismo que subyace a nivel de acciones, $g_\Lambda : A_1 \hookrightarrow A_2$, y de observaciones $g_\vartheta : B_1 \hookrightarrow B_2$ son inclusiones, entonces decimos que h es un morfismo de inclusión fuerte y lo escribimos de la forma $h : ob_1 \hookrightarrow ob_2$.*

En la Figura 3.3 se muestra dicha representación en la que E_1, E_2 representan trazas considerando sólo pasos, y V_1, V_2 representan trazas considerando sólo observaciones.

Aclaración de cada elemento de la Figura 3.3 (de izquierda a derecha):

- El objeto ob_1 está incluido *fuertemente* en ob_2.
- Las acciones del objeto ob_1 están incluidas en las de ob_2.
- Las trazas de pasos de ob_1 son una restricción de las de ob_2.

[1] A partir de este punto utilizaremos el símbolo $\hookrightarrow$ para denotar la inclusión, siendo el caso de la inclusión entre conjuntos la que, tradicionalmente, se escribe como $\subseteq$.

- El morfismo objetual es conmutativo.
- Las trazas de observaciones de ob_1 son una restricción de las de ob_2.
- Las observaciones del objeto ob_1 están incluidas en las de ob_2.

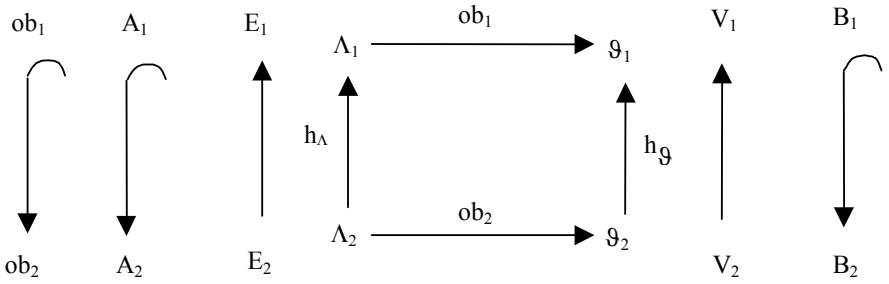

Figura 3.3: Diagrama del morfismo de inclusión fuerte.

Este morfismo de inclusión fuerte implica que el objeto ob_2 dispone de las acciones y observaciones de ob_1 de forma que las observaciones que se tengan de ob_1 son proyecciones de las observaciones de ob_2 y las trazas de ob_2 son enriquecimientos de las trazas de ob_1. La condición que impone el morfismo es que cualquier enriquecimiento de la vida $\lambda_1 \in \Lambda_1$ de ob_1, cuando es vista desde ob_2, equivale a la misma observación vista desde ob_1.

Por ejemplo, restringir una traza de un `computador` a sólo las acciones de la `memoria` y después observar tiene el mismo efecto que observar la traza del `computador` y restringir la atención únicamente a las observaciones relativas a la `memoria`. En este sentido, la `memoria` está encapsulada dentro del `computador`.

3.3. Inclusión observacional o débil

Si la idea de inclusión la trasladamos únicamente a la parte de observación y la relajamos para las acciones (fijando cualquier otro tipo de morfismos) se llega al concepto de morfismo de inclusión observacional que modela el que las acciones del entorno del objeto pueden afectar a las observaciones locales de forma directa.

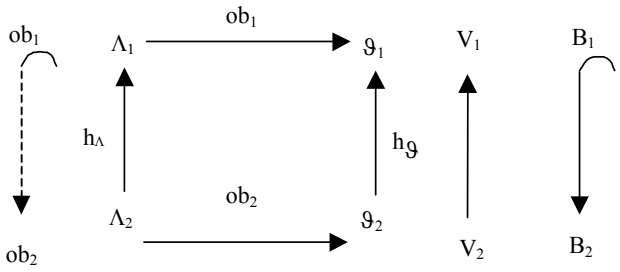

Figura 3.4: Diagrama del morfismo de inclusión débil.

Definición 17 ***Morfismo de inclusión observacional o débil.*** *Sea* $h : ob_1 \rightarrow ob_2$ *un morfismo entre objetos. Si el morfismo de observación* h_ϑ *es una inclusión y el morfismo a nivel de acciones* h_Λ *es un morfismo arbitrario, entonces se dice que* h *es un morfismo de inclusión observacional y lo escribimos como* $h : ob_1 \dashrightarrow ob_2$.

La Figura 3.4 explica la situación en la que V_1, V_2 representan las trazas incluyendo sólo observaciones.

Tal como en el caso del morfismo de inclusión fuerte, las observaciones sobre ob_1 son vistas de las observaciones de ob_2 pero, y esto es importante, a partir de los ciclos de vida de ob_2 no se derivan los de ob_1. Esto puede utilizarse para modelar inclusiones débiles de un objeto ob_1 dentro de otro objeto complejo ob_2 de manera que las acciones del objeto complejo afectan a las observaciones que sobre el objeto ob_1 se hagan, pero sólo a través de servicios locales al objeto incluido débilmente. Es decir, h_Λ expresa cómo las trazas *externas*[2] se llevan a cabo localmente. Esto debe indicarse expresando la relación o llamada[3] que existe entre acciones externas al objeto a las que son locales a él.

Consideremos el objeto `pila` que es incluido débilmente dentro del objeto `usuario:`

$$(\texttt{pila\_en\_usuario}) : \texttt{pila} \dashrightarrow \texttt{usuario}$$

Para cada acción de `pila`, tal como `pop`, `push`, etc., se asume que el `usuario` posee su correspondiente evento privado `call-pop`, `call-push`,

[2]Del resto de la sociedad de objetos, para ser más generales.
[3]Que posteriormente se definirá como *relación call* y *relación shared*.

etc. que llaman a las acciones de `pila`. De esta forma, el `usuario` utiliza sus acciones privadas para operar con `pila`, nunca directamente las acciones de `pila`. Así, la inclusión débil aparece al sustituir las llamadas a `pila` por las correspondientes acciones de `pila` y olvidando el resto de acciones del `usuario` sin hacer una correspondencia de cada traza del `usuario` a una traza de `pila`.

De este modo, varios objetos `usuario` pueden compartir la misma `pila` de forma que todos pueden operar sobre ella y todos observan los efectos de todas las operaciones, incluso las causadas por otros. Sin embargo, cada `usuario` puede definir atributos adicionales que cambian como resultado de sus llamadas a la `pila` y que, evidentemente, no pueden ser observados por el resto de los objetos de la clase `usuario`.

3.4. Agregación

Veamos qué repercusiones tiene la caracterización formal anterior desde el punto de vista del objeto complejo tal como se plantea en [6]. Un objeto puede ser visto como una cuádrupla $< A, Att, \Lambda, \vartheta >$ donde, A es el conjunto de acciones, Att el conjunto de atributos, Λ las trazas posibles y ϑ las observaciones posibles. La agregación por inclusión fuerte de dos objetos independientes $ob \equiv< A, Att, \Lambda, \vartheta >$ y $ob' \equiv< A', Att', \Lambda', \vartheta' >$ da lugar al objeto complejo que está definido por la cuatrupla $< A \oplus A' + A \times A', Att \oplus Att', \Lambda \parallel \Lambda', \vartheta + \vartheta' >$

Es decir, el espacio de acciones de la agregación es la unión disjunta de las acciones de los componentes[4] más el producto[5] de dichos alfabetos; el conjunto de atributos es la unión disjunta de los conjuntos de atributos[6] de los componentes; las trazas $\Lambda \parallel \Lambda'$ corresponderán a un entrelazado de las trazas de los componentes, y por último, las observaciones posibles son la suma de las observaciones de dichos componentes.

Dicho de otro modo, el conjunto de trazas posibles para el objeto agregado son las trazas tales que (con $\lambda \in \Lambda \parallel \Lambda'$, donde $\lambda \downarrow A$ denota la restricción de λ a las acciones de A.):

[4]Resaltamos que es así para la inclusión fuerte. Para la débil es un subconjunto de $A \oplus A' + A \times A'$

[5]El producto entre conjuntos permite obtener las combinaciones de acciones simultáneas vistas como nuevas acciones.

[6]Recordemos que el morfismo de observación es una inclusión tanto para la inclusión fuerte como para la débil.

$$\lambda \downarrow A \in \Lambda \wedge \lambda \downarrow A' \in \Lambda'$$

Las acciones del objeto agregado conllevan la ejecución concurrente de acciones de A y A'. Es decir, el objeto complejo agregado puede, en cada paso, llevar a cabo una acción de A (y ninguna de A'), o una de A' (y ninguna de A) o una tanto de A como de A'.

Definición 18 ***Relación Call.*** *Dado* $ob_1 \equiv < A_1, Att_1, \Lambda_1, \vartheta_1 >$ *un objeto y existiendo un morfismo de inclusión (fuerte o débil) conteniendo a otro objeto* $ob_2 \equiv < A_2, Att_2, \Lambda_2, \vartheta_2 >$*, con* $a_1 \in A_1$*,* $a_2 \in A_2$*, entonces decimos que* a_1 ***calls to*** a_2 *sii:*

$$\forall \lambda_1 \in \Lambda_1, \forall s \in \lambda_1 : a_1 \in s \Rightarrow a_2 \in s$$

Definición 19 ***Relación Shared.*** *Dado* $ob_1 \equiv < A_1, Att_1, \Lambda_1, \vartheta_1 >$ *un objeto y existiendo un morfismo de inclusión (fuerte o débil) conteniendo a otro objeto* $ob_2 \equiv < A_2, Att_2, \Lambda_2, \vartheta_2 >$*, con* $a_1 \in A_1, a_2 \in A_2$*, entonces decimos que* a_1 ***is shared with*** a_2 *sii:*

$$\forall \lambda_1 \in \Lambda_1, \forall s \in \lambda_1 : a_1 \in s \Leftrightarrow a_2 \in s$$

Resaltamos que mediante $s \in \lambda$ se denota la ocurrencia de un paso en alguna traza λ. De esta forma, las ocurrencias de a_1 sólo son posibles si el correspondiente a_2 ocurre en s. La *relación call* caracteriza la comunicación *síncrona* y *asimétrica*. La bidireccionalidad viene caracterizada por la *relación shared* como comunicación *síncrona* y *simétrica*. Ambas proveen una semántica para la sincronización entre objetos dentro de la definición de objetos complejos. A continuación se definirán las distintas posibilidades dentro de la agregación y las justificaremos considerando el marco formal introducido.

3.4.1. Formas de agregación

Históricamente, la agregación viene siendo la forma de representar que un objeto complejo tiene como componentes a otros objetos. Esta relación *parte_de* puede ser expresada desde varios puntos de vista dependiendo de la relación por morfismos que exista entre el objeto agregado y sus componentes.

Desde la tradición del modelado semántico hasta los lenguajes de especificación OO actuales, la definición de agregación varía en función de

los criterios tenidos en cuenta a la hora de fijar el significado de "poner juntos" determinados objetos para formar un objeto compuesto. La agregación en los entornos de programación orientados a objetos ha tenido que ser simulada mediante el uso de atributos "objeto-evaluados" que referencian a otros objetos.

Pensamos que es posible caracterizar los diferentes tipos de agregación de acuerdo con las siguientes dimensiones dentro del alcance de la jerarquía de la agregación.

- *¿Puede un objeto agregado compartir objetos componentes con otros objetos agregados*?

  Caso afirmativo tenemos objetos agregados **no disjuntos,** es decir, objetos agregados que pueden compartir componentes. En caso contrario, dichos objetos agregados son **disjuntos** pues no comparten componentes con otros objetos agregados.

- *¿La composición de un objeto agregado varía por la ocurrencia de acciones o bien está determinada de forma estática*?

  Aquellos cuya composición está afectada por acciones se llaman objetos agregados **dinámicos**, y aquellos cuya composición queda fija desde su creación se llaman objetos agregados **estáticos**. La agregación dinámica significa que se puede alterar la composición de los objetos mediante la invocación de acciones de inserción y borrado (que se suponen implícitamente definidas para cada objeto agregado dinámico).

- *¿Puede un objeto de una clase componente existir sin ser componente de un objeto agregado*?

  Caso afirmativo, la agregación se dice que es **flexible**, en otro caso decimos que es **estricta**.

- *¿Cuántos objetos componentes de una clase componente puede tener asociados un objeto agregado*?

  Diremos que la agregación es **univaluada** para el caso mínimo y **multivaluada** para el caso de valores mayores que uno.

- *¿Puede el objeto agregado no tener objetos de una clase componente en algún instante*?

  Dependiendo de la respuesta, el componente será con **nulos** permitidos o bien con **no nulos**.

Hablando en términos de cardinalidad mínima y máxima, tenemos las correspondencias indicadas en las siguientes tablas:

- Cardinalidad desde la clase agregada a la clase componente:

mínima cardinalidad	0	**nulos**
	$\geqslant 1$	**no nulos**
máxima cardinalidad	1	**univaluados**
	>1	**multivaluados**

- Cardinalidad desde la clase componente a la clase agregada

mínima cardinalidad	0	**flexible**
	$\geqslant 1$	**estricta**
máxima cardinalidad	1	**disjunta**
	>1	**no disjunta**

Además, dependiendo del tipo de inclusión entre el objeto agregado y el objeto componente también distinguiremos entre agregación **inclusiva** y agregación **relacional** (o referencial). En la agregación inclusiva cualquier objeto componente está completamente encapsulado en la agregación en el sentido de que sólo le pueden ocurrir acciones locales, adecuadamente coordinadas con las acciones del objeto agregado.

3.4.2. Formas de agregación y aspectos formales

Es importante tener en cuenta la relación existente entre los aspectos formales presentados y los distintos tipos de agregación. Cada forma de agregación está referida a propiedades del componente, así, para una agregación concreta, cada componente presenta unas características frente al agregado, dando las características de todos los componentes en conjunto. La presentación es intuitiva pues una justificación formal queda fuera del alcance de este libro.

- *Componente Inclusivo/Relacional*

La agregación *inclusiva* implica hacer uso de la inclusión fuerte definida anteriormente mientras que la agregación *relacional* tiene su justificación en la inclusión débil.

- *Agregación Dinámica/Estática*

Si un objeto de una clase C_1 es componente de otro objeto de una agregada C_0 de forma *estática* entonces permanecerá como componente desde la creación del objeto agregado y hasta la destrucción del mismo (si ocurre). Será *dinámica* en caso contrario.

- *Componente con Nulos/sin Nulos*

Si una clase C_1 es componente de otra agregada C_0 de forma *sin nulos* entonces para toda instancia de C_0 debe existir al menos una instancia de C_1 tal que ésta última esté incluida (fuerte o débilmente) en dicha instancia de C_0. Será *con nulos* si no existe dicha restricción.

- *Componente Univaluado/Multivaluado*

Si una clase C_1 es componente de otra agregada C_0 de forma *univaluada* y una instancia de C_1 está incluida (fuerte o débilmente) en alguna instancia ob_0 de C_0 entonces, no puede existir otra instancia de C_1 que también esté incluida en ob_0 al mismo tiempo. Será *multivaluada* si no existe dicha restricción. Esta definición también puede verse de forma análoga desde el agregado hacia el componente.

- *Componente Estricto/Flexible*

Si una clase C_1 es componente de otra agregada C_0 de forma *estricta* entonces toda instancia de C_1 estará incluida (fuerte o débilmente) en alguna de C_0. Será *flexible* si no existe dicha restricción.

- *Componente Disjunto/No Disjunto*

Si una clase C_1 es componente de una clase agregada C_0 de forma *disjunta* entonces una instancia de C_1 que esté incluida (fuerte o débilmente) en alguna instancia de C_0 no podrá estar incluida en otra instancia de C_0, dentro del mismo componente. Será *no disjunta* en el caso opuesto.

3.5. Herencia

Las clases tienen un tipo concreto que caracteriza el conjunto de todas las instancias posibles de la misma. La **herencia** es una característica importante del paradigma objetual que describe cómo una clase hereda propiedades desde otra. En términos generales, la clase que hereda dispone de las propiedades de otra clase, y puede añadir o cambiar algunas de estas propiedades. La herencia se establece mediante relaciones de **especialización** o de **rol** como mecanismos de modificación incremental a través de relaciones de orden entre los tipos o plantillas de clases. El proceso inverso a la especialización es la **generalización**.

Definición 20 ***Subclases y superclases.*** *La plantilla de una subclase se define basándose en modificaciones sobre las plantillas de otras clases llamadas superclases y a partir de las cuales hereda.*

El modelo OASIS considera tres tipos de estructuras taxonómicas ortogonales entre sí:

- *Subclases Estáticas.*
- *Subclases Dinámicas.*
- *Subclases de Rol.*

La relación que existe, por ejemplo, entre instancias de las clases `mujer` y `persona` es tal que una instancia de `mujer` es siempre una instancia de `persona` siendo entonces una subclase **estática**.

La migración entre clases es un fenómeno por el cual un objeto puede cambiar entre clases a lo largo de su existencia. Tal como se propone en [34] consideraremos dos casos en los cuales puede haber migración entre clases: **herencia dinámica** y **herencia por rol**. Un ejemplo clásico que puede modelarse de dos formas es el que considera una `persona` que llega a ser `empleado`. Podemos hacer que `empleado` sea una subclase **dinámica** de `persona`. Así, cada instancia de `empleado` es idéntica a una instancia de `persona`, es decir, posee el mismo *oid* que la correspondiente instancia de `persona`. Como consecuencia, si contamos n `empleados` en un conjunto dado entonces dicho conjunto también poseerá n `personas`.

Por otro lado, si queremos modelar la situación por la que una instancia de `persona` puede llegar a ser distintas instancias de `empleado` (tanto simultáneamente como secuencialmente) entonces tendremos que dar a cada instancia de `empleado` su propio *oid*, distintos entre ellos y de los de `persona`. En este caso decimos que `empleado` es una subclase de **rol** de `persona` y a las instancias de `empleados` se les llama **roles**. Dado que los roles poseen sus propios *oids*, contar `empleados` no será igual que contar `personas`. Cuando contamos `empleados` realmente contamos cuántas `personas` han pasado al estado de ser un `empleado` (pueden existir "pluriempleados").

3.5.1. Clasificación de objetos

Definición 21 ***Instancia de una clase.*** *Una clase tiene asociado un conjunto potencial de objetos que llamaremos instancias de la clase.*

Definición 22 ***Clase de objetos y de Rol.*** *Si las instancias son objetos entonces la clase se llama clase de objetos, si las instancias son roles, entonces la clase se llama clase de rol. Tanto los objetos como los roles son vistos como objetos individuales que poseen estado y comportamiento.*

Para cada clase C, se distinguen los siguientes aspectos:

- La **intensión** de una clase, $int(C)$, es el conjunto de *todas* las propiedades que son compartidas por todas las instancias de la clase. Representa la plantilla o tipo de la clase.

- La **extensión** de una clase, $ext(C)$, es el conjunto de *todas* las *posibles*[7] instancias de la clase.
- Dado un instante cualquiera t del sistema, el **conjunto existencia** (**población**) de una clase, $ext_t(C)$, es el conjunto de todas las instancias de la clase que existen en dicho instante.

En un instante concreto t, el conjunto total de objetos existentes en el sistema se denotará por $exists_t$. Para cualquier clase C se cumplirá que:

$$ext_t(C) = ext(C) \cap exists_t$$

Siendo C_1 y C_2 dos clases, si $ext_t(C_1) \subseteq ext_t(C_2)\ \forall t$, entonces diremos que C_1 es subclase, estática o dinámica, de C_2, y lo escribiremos como C_1 *is_a* C_2,

Considerando la definición de ext, si esto se cumple, se tendrá que $ext(C_1) \subseteq ext(C_2)$. Teniendo $ext(C_1) \subseteq ext(C_2)$, existe una relación de inclusión inversa entre las intensiones de C_1 y de C_2. Es decir, si $ext(C_1) \subseteq ext(C_2)$, entonces $int(C_2) \subseteq int(C_1)$.

Ejemplo 11 `coche` *is_a* `vehículo` *pues* $ext($`coche`$) \subseteq ext($`vehículo`$)$*, lo que es equivalente a decir que* $int($`vehículo`$) \subseteq int($`coche`$)$*. La relación de inclusión entre las intensiones representa la relación de herencia de la superclase a la subclase.*

La relación de inclusión existente entre las extensiones se corresponde con el morfismo de inclusión existente entre el conjunto de identificadores de los objetos de la superclase y la subclase. Desde el punto de vista del comportamiento, se supone implícito un morfismo que preserva el comportamiento entre las clases relacionadas por la herencia. Así, si `cliente` es una subclase de `persona` entonces todos los posibles ciclos de vida del `cliente` dan mediante restricciones (proyecciones), ciclos de vida de `persona`. El diagrama que definimos conmutativo al hablar de agregación (ver Figura 3.2) lleva la idea de que *observar* + *proyectar* sea equivalente a *proyectar* + *observar* para cualquier relación que incluya comportamiento y estado (relaciones *is-a*, *part-of*, *as-a*, etc.).

Los diferentes mecanismos de herencia que se definen en este capítulo determinan propiedades distintas para los morfismos asociados tanto a los mecanismos de identificación como al comportamiento.

[7] Es importante precisar este adjetivo para la definición.

3.5.2. Particiones estáticas

Definición 23 ***Partición Estática.*** *Siendo $C_1, ..., C_n$ subclases de otra clase C_0, decimos que $\{C_1, ..., C_n\}$ es una partición estática de C_0, y se denota $\{C_1, ..., C_n\} \overset{is\_a}{\rightarrow} C_0$, si se cumple que:*

$$ext(C_0) = \cup ext(C_i), \forall i = 1, ..., n$$
$$ext(C_i) \cap ext(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n$$

Podemos particularizar en cualquier instante t del sistema:

$$ext_t(C_0) = \cup ext_t(C_i), \forall i = 1, ..., n$$
$$ext_t(C_i) \cap ext_t(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n$$

Existe una cardinalidad implícita asociada a cada partición por la que cada instancia de la subclase está relacionada con exactamente una instancia de la superclase y cada instancia de la superclase está relacionada exactamente con una instancia de la subclase. Así, si o es una instancia de C_0, entonces es instancia de exactamente una subclase de cada partición[8] existente.

Cada nueva partición de una clase implica un número mayor de clases como consecuencia del producto cartesiano entre las subclases de las particiones.

Definición 24 ***Subclase estática.*** *Una subclase estática pertenece a una partición estática o a una intersección de subclases estáticas. A partir de todas las posibles combinaciones de todas las particiones estáticas obtenemos un conjunto de subclases estáticas de más bajo nivel con las propiedades:*

- Son disjuntas entre ellas.
- Cubren todas las posibles situaciones del objeto de la clase particionada.
- No tienen más particiones estáticas asociadas.

Definición 25 ***Especies.*** *Las clases de más bajo nivel se llaman especies [34].*

[8]Nótese que se pueden definir varias particiones para una misma clase.

El concepto de especie aporta ventajas de tipo metodológico dado que el modelo se hace más claro y reduce las posibilidades de error. Por otra parte, es más fácil especificar la dinámica del modelo, dado que debemos asociar los eventos de creación sólo para las especies y nunca para el resto de las clases.

Ejemplo 12 *Para la clase* `vehículo` *podemos considerar las siguientes particiones estáticas:*

`{camión, coche, otro_vehículo}` $\stackrel{is\_a}{\rightarrow}$ `vehículo`
`{gasolina, diesel, otro_tipo}` $\stackrel{is\_a}{\rightarrow}$ `vehículo`

Ejemplo 13 *Algunas especies que se generan para el ejemplo anterior, son las siguientes:*

`coche*diesel, coche*gasolina, camión*diesel, camión*gasolina,` etc.

3.5.3. Particiones dinámicas

Definición 26 ***Partición dinámica.*** *Siendo $C_1, ..., C_n$ subclases de otra clase C_0, decimos que $\{C_1, ..., C_n\}$ es una partición dinámica de C_0, y se denota $\{C_1, ..., C_n\} \stackrel{is\_a}{\mapsto} C_0$, si para cada instante t se cumple que:*

$$ext_t(C_0) = \cup ext_t(C_i), \forall i = 1, ..., n \tag{3.1}$$

$$ext_t(C_i) \cap ext_t(C_j) = \emptyset, \forall i \neq j, \forall i, j = 1, ..., n \tag{3.2}$$

Sin embargo, puede haber instantes del sistema t_1 y t_2, con $t_1 \neq t_2$, y también i, j con $i \neq j$, tales que:

$$ext_{t_1}(C_i) \cap ext_{t_2}(C_j) \neq \emptyset \tag{3.3}$$

Y por lo tanto, se tendrá que:

$$ext(C_i) = ext(C_j), \forall i, j = 0, ...n \tag{3.4}$$

Las fórmulas 3.1 y 3.2 son idénticas a las que satisfacen las particiones estáticas pero particularizadas a instantes dados. La fórmula 3.3 implica que pueden existir al menos dos instantes distintos tal que al cambiar del uno al otro, al menos un objeto "se mueve" desde una clase a la otra. La fórmula 3.4 es consecuencia de que, en principio, cualquier objeto puede llegar a estar en cualquiera de las clases de la partición y la igualdad se mantiene también para la superclase C_0. El ejemplo anterior de la partición estática implicaque una instancia de la clase `camión` no puede pasar a serlo desde la clase `coche`, o viceversa. Esto no es así en las particiones dinámicas tal como veremos a continuación.

Definición 27 ***Subclase dinámica.*** *Una subclase dinámica es una subclase en una partición dinámica o bien, es una intersección entre una subclase dinámica y otra clase. Si C_1 es una subclase dinámica de C_2, escribimos*[9] *$C_1 \overset{is\_a}{\mapsto} C_2$, y diremos que C_1 es una especialización dinámica de C_2.*

No se permite el particionar una subclase dinámica en particiones estáticas, es decir, no será posible que se cumpla $C_1 \overset{is\_a}{\rightarrow} C_2 \wedge C_2 \overset{is\_a}{\mapsto} C_3$. Esta restricción facilita el proceso de formalización y no excluye casos importantes de modelado.

Ejemplo 14 *Para la clase* ***`vehículo`*** *podemos considerar la siguiente partición dinámica:*

$$\texttt{\{funcionando, estropeado\}} \overset{is\_a}{\mapsto} \texttt{vehículo}$$

En cada estado posible, la existencia de un objeto `vehículo` se particiona en la existencia de estar `funcionando` o `estropeado` por lo que una instancia de `vehículo` es alguno de los dos. Sin embargo, un objeto podría pasar de `funcionando` a `estropeado` y viceversa.

Ejemplo 15 *Considerando la partición estática del* ***`vehículo`****, algunas especies que se generan son las siguientes:*

[9] Nótese que se ha diferenciado de la estática en el tipo de flecha utilizada.

```
funcionando*coche*gasolina,
funcionando*camión*diesel,
estropeado*coche*diesel,
estropeado*camión*diesel,...
```

3.5.4. Migración entre clases

Lo que normalmente se conoce como que un objeto *migre de una clase a otra* debe modelarse como un objeto que *cambia de una subclase a otra* en alguna partición dinámica. Consideremos el ejemplo en el que una `persona` puede ser o no, en cualquier momento de su existencia, `estudiante`:

$$\texttt{\{estudiante, no\_estudiante\}} \overset{is\_a}{\mapsto} \texttt{persona}$$

El evento `llegar_a_ser_estudiante` debe ocurrir en la vida de la clase `persona` y no en la de `estudiante` pero `estudiante` hereda este evento, lo que resulta en una contradicción. El hecho de que la partición de la clase en las subclases dinámicas sea completa soluciona dicho problema.

Instancias de la clase `no_estudiante` tienen, en general, las mismas propiedades que `persona` y la propiedad adicional de que pueden pasar a ser `estudiante` (propiedad que `estudiante` no posee). Igualmente, objetos que sean instancias de la clase `estudiante` tienen la propiedad adicional de que pueden pasar a ser `no_estudiante` (propiedad que, de forma análoga, no posee `no_estudiante`).

Una diferencia importante entre las subclases dinámicas y las estáticas es que en las primeras el conjunto *existencia* de la subclase puede cambiar sin que cambie en la superclase. En cambio, si el conjunto de existencia cambia en una subclase estática entonces cambia también en el conjunto de existencia de la superclase (por ejemplo, como consecuencia de suprimir un objeto). Así, en el ejemplo de la subclase estática `vehículo` la creación de una instancia en `coche` es también la creación en `vehículo`.

3.5.5. Roles

El concepto de rol asociado al paradigma orientado a objetos fue introducido en el modelo ORM (Object with Role Model) [25]. El uso de

roles asociados a los objetos permite la representación de conductas diferentes de un objeto y la evolución dinámica de dicha conducta. Una clase puede tener asociada diferentes subclases de rol, cada uno representando un patrón de conducta específico para un objeto de dicha subclase. El objeto sigue siendo instancia de una sola clase pero puede desempeñar distintos roles a lo largo de su existencia. De esta forma puede representarse la evolución temporal del comportamiento de un objeto. Las instancias de una subclase de rol pueden ser creadas o destruidas. Un objeto puede estar asociado al mismo tiempo a dos o más instancias de rol, y por supuesto, puede desempeñar varias instancias de la misma subclase de rol simultáneamente.

Definición 28 ***Rol.*** *Un rol es como un objeto excepto que tiene una relación especial con otros objetos o roles que son los que desempeñan el rol. Un rol puede ser desempeñado por un objeto, o por otro rol (y que denominamos player).*

Consideraremos la función *played_by* en el modelo, tal que si R es un conjunto de clases de rol, entonces existe una clase de objetos (o de rol) tales que en cada instante t se cumple que:

$$played\_by : ext_t(R) \rightarrow ext_t(P)$$

donde P representa la clase de los objetos que desempeñan el rol. Entonces, $\forall r \in ext_t(R)$, $played\_by(r)$ es quien desempeña el rol para r. Además, siempre se cumplirá que:

- Existe exactamente un $played\_by(r)$.
- La existencia de r está supeditada a la existencia de $played\_by(r)$.
- Pueden existir varios roles para un mismo *player*, aun cuando estos roles sean instancias de la misma clase de rol.

La **delegación** es un concepto particularmente útil para expresar la compartición de propiedades. El envío de mensajes es distinto del reuso de la conducta asociado a la herencia[10]. Mediante delegación, determinados objetos son compartidos a través de invocaciones. De esta forma

[10]En un sentido estricto, el mecanismo de delegación es suficiente para simular la herencia.

es posible definir delegación de roles a *players*. Por ejemplo, supongamos que queremos modelar un `empleado` e como el rol de una `persona` p y `edad` es un atributo de `persona` y no de `empleado`. Entonces, `edad`(e) debería ser un error. Podemos recuperar el error mediante delegación de la evaluación de `edad` a $played\_by(e)$. Es decir, pasamos de tener `edad`(e) a tener `edad`$(played\_by(e))$. Evidentemente, la delegación también puede definirse para las acciones.

En general, cada clase (incluyendo una de rol) puede especializarse en uno o más conjuntos de subclases de rol, llamados **grupos de rol**. Cada grupo de rol representa un conjunto de subclases de rol mutuamente exclusivas, es decir, de forma simultánea un *player* puede desempeñar roles de como máximo una clase de rol dentro de cada grupo de rol. A diferencia de las particiones tipo is_a, puede haber instancias de un *player* que no desempeñen ningún rol en un grupo de rol, es decir, un grupo de rol puede contener una sola clase de rol. Es importante destacar la posibilidad de definir particiones estáticas y dinámicas de las clases de rol. Por otro lado, la definición de roles y de particiones no puede ser cíclica.

Es importante destacar que las subclases heredan tanto las propiedades de la superclase como los roles que se pueden desempeñar. Para el ejemplo, una instancia de la clase `estudiante` puede desempeñar el rol de `empleado`. Por otro lado, una clase de rol puede particionarse tanto de forma dinámica como estática (tal como se ha hecho en el ejemplo con la clase `empleado`).

Ejemplo 16 *Detallamos a continuación una combinación de particiones estáticas, dinámicas y de rol en la misma jerarquía:*

$$\texttt{empleado} \overset{played\_by}{\longrightarrow} \texttt{persona}$$
$$\texttt{\{estudiante, no\_estudiante\}} \overset{is\_a}{\mapsto} \texttt{persona}$$
$$\texttt{\{temporal, permanente\}} \overset{is\_a}{\mapsto} \texttt{empleado}$$
$$\texttt{\{hombre, mujer\}} \overset{is\_a}{\rightarrow} \texttt{persona}$$
$$\texttt{director} \overset{played\_by}{\longrightarrow} \texttt{permanente}$$

3.5.6. Herencia múltiple

Definición 29 ***Herencia múltiple.*** *La herencia múltiple establece que las instancias de una clase heredan propiedades a partir de dos o más*

superclases. En nuestro contexto, cada especie que involucre a más de una clase es ya una clase con herencia múltiple.

Ejemplo 17 *Las especies* `funcionando*coche` *y* `coche*diesel` *son casos de herencia múltiple. El conjunto de propiedades asociadas a dichas clases es la suma de las propiedades de las clases involucradas.*

Cualquier propiedad emergente que se desee añadir a las consideradas por defecto debe ser definida en la plantilla de dicha clase.

3.5.7. Ciclos de vida en la herencia *is_a*

Los objetos están sujetos a restricciones que limitan su comportamiento al considerar distintos estados. Parte de estas restricciones pueden representarse mediante *diagramas de ciclo de vida del objeto* [34].

Un concepto de subclasificación similar al expuesto anteriormente es el *Principio de Sustitutibilidad* [31]. Dicho principio establece que si C_1 *is_a* C_0 entonces el ciclo de vida de las instancias de C_1 debería ser tal que una instancia de C_1 pueda usarse en *cualquier contexto* en el que una instancia de C_0 pueda usarse. Son varios los enfoques propuestos que satisfacen este principio. Un criterio comúnmente aceptado es que el ciclo de vida de una instancia de la clase `coche` (con la definición anterior) es una especialización válida del ciclo de vida de `vehículo` dado que a partir de una traza de `coche` si se restringe únicamente a los eventos definidos en `vehículo` obtenemos una traza válida para la superclase, es decir, `vehículo`.

Para el caso de las particiones dinámicas, en cada instante una instancia de la superclase está en alguna de las subclases y puede migrar a cualquier otra si cumple las restricciones oportunas. El ciclo de vida de la superclase describirá la forma en la que sus instancias transitan entre las clases de la partición dinámica. Las subclases estáticas heredan el diagrama del ciclo de vida de la superclase y modifican su comportamiento (añadiendo nuevas propiedades) pero siempre compatible en cuanto a comportamiento. De esta forma, en una jerarquía de clases estática el ciclo de vida más complejo se encuentra en las hojas de la jerarquía. En cambio, en la partición dinámica el ciclo de vida más complejo se encuentra en la raíz de dicha jerarquía. *Esto es así dado que las particiones estáticas particionan el espacio de objetos mientras que las dinámicas particionan el espacio de estados.*

Capítulo 4

Interacción entre objetos

En este capítulo se introduce la noción de interacción entre objetos. En OASIS, un sistema de información es una sociedad de objetos autónomos, concurrentes y que interactúan entre sí. La interacción entre objetos está basada en comunicación. Comenzaremos presentando algunos conceptos relacionados que se utilizarán para describir los mecanismos de comunicación ofrecidos por OASIS.

4.1. Identificación de objetos

Definición 30 ***Oid (object identifier)****. Cada objeto posee un oid. El oid establece la identidad del objeto y tiene las siguientes características:*

- Constituye un identificador único y global para cada objeto dentro del sistema.
- Es determinado en el momento de la creación del objeto.
- Es independiente de la localización física del objeto, es decir, provee completa independencia de localización.
- Es independiente de las propiedades del objeto, lo cual implica independencia de valor y de estructura.
- No cambia durante toda la vida del objeto. Además, un *oid* no se reutiliza aunque el objeto deje de existir.

- No se tiene ningún control sobre los *oids* y su manipulación resulta transparente.

Sin embargo, en el ámbito de la especificación y operación del sistema, es preciso contar con algún medio para hacer referencia a un objeto en particular.

Definición 31 ***Mecanismo de identificación.*** *Siendo Oids el conjunto de oids de objetos del sistema, C una clase y $Oids_c$ el conjunto de oids de objetos de la clase C, entonces $Oids_c \subseteq Oids$. Además, siendo Att el conjunto de atributos de la clase C y $Att_{cons} \subseteq Att$ un conjunto de atributos constantes de la clase C, si $\{att_1, att_2, ..., att_n\} \subseteq Att_{cons}$ y los sorts respectivos son $s_1, s_2, ..., s_n$ entonces llamaremos mecanismo de identificación a la función:*

$$id : s_1 \times s_2 \times ... \times s_n \rightarrow Oids_c$$

La función *id* establece una correspondencia entre una tupla de valores de atributos constantes y el *oid* de un objeto de la clase C, con $oid \in Oids_c$. La tupla representa el concepto de clave candidata.

Definición 32 ***Valor de un mecanismo de identificación.*** *Siendo id un mecanismo de identificación, entonces llamamos valor de un mecanismo de identificación a $\underline{id}$. Así, $\underline{id} = \theta\, id$ y θ es una sustitución básica de los valores de atributos constantes que definen al mecanismo de identificación.*

Los mecanismos de identificación no son redundantes y permiten hacer referencia a objetos en el ámbito del problema, utilizando propiedades del objeto. Sin embargo, internamente la referencia e identificación del objeto es mediante su *oid*.

Un objeto puede tener más de un mecanismo de identificación aunque cada vez que se hace referencia a un objeto se utiliza sólo uno de ellos.

A partir de este punto se introducirán parcialmente aspectos de la sintaxis de OASIS para hacer más sencilla y precisa la exposición del capítulo. En el Apéndice B se adjunta la sintaxis completa del lenguaje OASIS.

Sintaxis

<def_identificador>	::=	<id_identificador> ':' (<lista_id_atributo>)

Cada atributo que define el mecanismo de identificación (en la lista <id_atributo>) es un atributo constante.

Ejemplo 18 *Una definición de tres mecanismos de identificación para los objetos de la clase* `despacho` *en un edificio.*

```
ocupante:(nombre);
nro_ext_telefonica:(nro_ext_telefonica);
ubicacion:(edificio, planta, puerta);
```

Cuando un mecanismo de identificación se define a base de sólo un atributo, puede usarse para denotarlo el mismo nombre del atributo.

4.2. Referencias a cliente y servidor

La necesidad de hacer referencia a otros objetos (de la clase que se está especificando o de otra clase) está estrechamente ligada a la interacción entre objetos. En este apartado nos centraremos en cómo hacer referencia a objetos en la especificación de una clase, sean clientes o servidores de las acciones que acontecen.

Sintaxis

<ref_objeto>	::=	**self**
	|	<clase> **[**<identificación>**]**
<identificación>	::=	'(**everyone**)'
	|	'(**someone**)'
	|	'**(**'<id_identificador>'**,**'
		'**[**'<lista_valor_atributo>'**]**' '**)**'
<valor_atributo>	::=	<expresión> | '**_**'

Observaciones

- Cuando se desea hacer referencia al mismo objeto que se está especificando se utilizará la palabra reservada **self**.

- Para hacer referencia a cada uno de los objetos de la clase se utiliza **(everyone)**.

- Si se quiere referenciar a un objeto de una clase (sin especificar uno en concreto) se utiliza **(someone)**.

- Para referirse a un conjunto de objetos que coinciden en alguno de los valores de la lista de valores de atributo se indicará el mecanismo de identificación y la lista de valores de atributos llevará los valores que deben coincidir en las posiciones adecuadas. Se utiliza '_' para señalar que se admite cualquier valor. Si todas las posiciones tienen un '_' el efecto es el mismo que usar **(everyone).**

Ejemplo 19 *El objeto de la clase* `despacho` *cuyo ocupante es Juan Pérez.*

```
despacho(ocupante,[''Juan Perez''])
```

Ejemplo 20 *El objeto de la clase* `despacho` *cuyo número de extensión telefónica es 3571.*

```
despacho(nro_ext_telefonica,[3571])
```

Ejemplo 21 *El objeto de la clase* `despacho` *ubicado en el edificio EUI, tercera planta y puerta A3.*

```
despacho(ubicacion,[''EUI'',3,''A3''])
```

Ejemplo 22 *Un objeto cualquiera de la clase* `despacho`*.*

```
despacho(someone)
```

Ejemplo 23 *Cada uno de los objetos de la clase* `despacho`*.*

```
despacho(everyone)
```

Ejemplo 24 *Los objetos de la clase* `despacho` *ubicados en el edificio de la EUI.*

```
despacho(ubicacion,[''EUI'',_,_])
```

4.3. Acciones

El concepto de acción incluye tres elementos: referencia del cliente, referencia del servidor y descripción del servicio. Así, una acción está definida por la tupla $\langle Cliente, Servidor, Servicio \rangle$. El cliente requiere un servicio a un servidor que ofrece dicho servicio. El servicio puede ser un evento o una operación. Las referencias del cliente y del servidor se definen según lo explicado en el apartado anterior. El servicio es un término instanciado mediante una sustitución de valores adecuados en sus parámetros.

Ejemplo 25 *Una acción cuyo cliente es un objeto de la clase* ***usuario*** *y cuyo servidor es un objeto de la clase* ***cuenta****, siendo* ***depósito(15000)*** *el servicio requerido.*

```
<usuario(nombre,[''Juan Perez'']),

            cuenta(numero,[100]),deposito(15000)>
```

Ejemplo 26 *Una acción cuyo cliente y servidor es el mismo objeto, donde* ***pagar_comisión*** *es el servicio requerido.*

```
<self,self,pagar_comision>
```

Ejemplo 27 *Una acción cuyo cliente es un objeto de la clase* ***administrador*** *y cuyo servidor son todos los objetos de la clase* ***artículo****, siendo* ***aumentar_precio(5)*** *el servicio requerido.*

```
<administrador(nombre,[''Maria'']),

            articulo(everyone),aumentar_precio(5)>
```

Ejemplo 28 *Una acción cuyo cliente es un objeto de la clase* ***goloso*** *y cuyo servidor es un objeto de la clase* ***máquina*** *(máquina de chocolatinas), siendo* ***cancelar*** *el servicio solicitado.*

```
<goloso(nombre,[''Juan Perez'']),

            maquina(numero,[10]),cancelar>
```

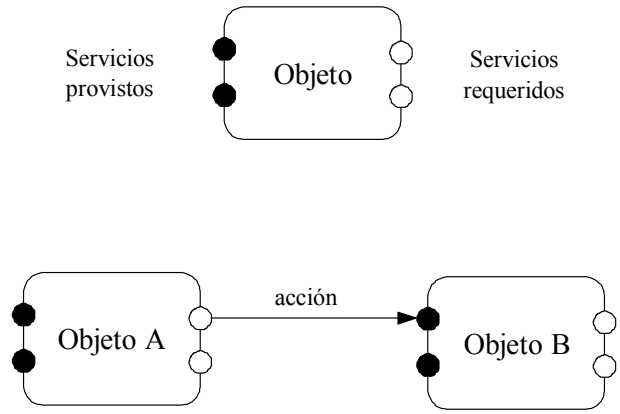

Figura 4.1: Perspectivas cliente y servidor de un objeto.

Ejemplo 29 *Una acción cuyo cliente es un objeto cualquiera de la clase* `goloso` *y cuyo servidor es un objeto de la clase* `máquina` *(máquina de chocolatinas), siendo* `introducir_moneda` *el servicio solicitado.*

```
<goloso(someone),maquina(numero,[10]),introducir_moneda>
```

Nótese que al establecer el servicio de una acción no se hace distinción entre requerir un evento o requerir una operación. Esto constituye una ventaja puesto que el cliente no tiene por qué saber detalles del servicio que requiere.

La Figura 4.1 muestra cómo un objeto tiene una perspectiva cliente y una perspectiva servidor según esté requiriendo u ofreciendo servicios, respectivamente. Modelar la perspectiva cliente significa tener la posibilidad de reflejar en la especificación de la clase cliente el hecho de que "**todo servicio requerido también es un evento para el objeto cliente**". Ambas perspectivas son tratadas uniformemente bajo la noción de acción antes descrita.

En la especificación de una clase, las acciones asociadas a servicios provistos o requeridos se incluyen en la definición de fórmulas (precondiciones, evaluaciones y disparos) y de procesos (operaciones y protocolos).

Las acciones en su formato completo ($\langle Cliente, Servidor, Servicio \rangle$) pueden dificultar la lectura y edición de la especificación. Para evitar este inconveniente se pueden utilizar las siguientes simplificaciones:

a) Cuando se trata de un servicio ofrecido por el objeto (el propio objeto es el servidor) la acción se escribirá como **Cliente:Servicio.** Si el cliente no es relevante se escribirá sólo **Servicio.**

b) Cuando se trata de un servicio solicitado por el objeto (el propio objeto es el cliente) la acción se escribirá como **Servidor::Servicio.**

c) En el caso particular en el cual el cliente y el servidor son el mismo objeto se debe diferenciar entre la acción de requerir el servicio y la acción de proveerlo. En este caso se escribirá **::Servicio** para representar la acción por requerir y **self:Servicio** para representar la acción por proveer.

Sintaxis

<acción>	**::=**	[<cliente>] [<servidor>] <servicio>
<cliente>	**::=**	<ref_objeto> '**:**'
<servidor>	**::=**	<ref_objeto> '**::**' | '**::**'

Ejemplo 30 *En la clase* ***cuenta****, una evaluación asociada a la acción cuyo cliente es un objeto cualquiera y cuyo servidor es el propio objeto de la clase* ***cuenta****, siendo el servicio* ***depósito(Cantidad:int)****.*

```
saldo=S [deposito(Cantidad)] saldo=S+Cantidad;
```

"Cuando `saldo=S` y ocurre la acción cuyo evento es `depósito(Cantidad)` inmediatamente después se cumple que `saldo=S+Cantidad`"

Ejemplo 31 *En la clase* ***cuenta****, una parte de una expresión de proceso (supondremos que se trata de una operación) en la cual el cliente y servidor de una acción corresponden al objeto de la clase* ***cuenta*** *(la clase que se está especificando), siendo el servicio* ***pagar_comisión****.*

```
... AJUSTAR4= ::pagar_comision.AJUSTAR5; ...
```

"Cuando el estado del proceso es `AJUSTAR4`, debe ocurrir la acción cuyo cliente y servidor es el objeto de la clase especificada (`cuenta`). El evento de la acción es `pagar_comisión`. Después de ocurrida la acción el estado del proceso es `AJUSTAR5`".

Ejemplo 32 *En la clase* ***administrador****, una obligación de ejecutar la acción cuyo cliente es un objeto de la clase* ***administrador*** *y cuyo servidor son todos los objetos de la clase* ***artículo****, siendo el servicio* ***aumentar_precio(Porcentaje:int)****.*

```
ventas > 1000000
        [¬articulo(everyone)::aumentar_precio(5)] false
```

"Al satisfacerse `ventas > 1000000` debe ocurrir una acción cuyo cliente es el objeto de la clase especificada (`administrador`), el servidor es cada objeto de la clase `artículo` y el servicio es `aumentar_precio(5)`".

Ejemplo 33 *En la clase* ***máquina****, una precondición para una acción cuyo cliente es el objeto de la clase* ***goloso*** *"Juan Pérez" y cuyo servidor es el mismo objeto (máquina de chocolatinas), siendo* ***chocolatina*** *el servicio solicitado.*

```
¬(numero_chocs⩾10)
        [goloso(nombre,[''Juan Perez'']):chocolatina] false
```

"Se prohíbe que cuando el número de chocolatinas (`numero_chocs`) es menor que 10 acontezca el servicio `chocolatina` requerido por el objeto de la clase `goloso` cuyo nombre es Juan Pérez".

Nótese que en la especificación de una clase, el servicio de una acción es siempre un servicio de la clase servidora. Es decir, a menos que la clase servidora sea **self**, dicho servicio no está en la signatura de la clase cliente. Sin embargo, en el ámbito de una acción vista como una tupla $\langle Cliente, Servidor, Servicio \rangle$, tanto las acciones recibidas como las acciones enviadas pertenecen a la signatura de la clase especificada.

4.4. Interfaces

Las interfaces son el mecanismo usado en OASIS para establecer canales de comunicación entre clientes y servidores. Cuando un objeto es declarado cliente de un determinado servidor se está definiendo una relación de accesibilidad entre ambos. En este sentido, las interfaces actúan como mecanismo de seguridad. De esta forma, un objeto sólo puede requerir a la sociedad de objetos los servicios que le son accesibles según las definiciones de interfaz en las cuales él es cliente.

En OASIS, por simplicidad, sólo se provee de interfaces de proyección, siendo éstas las de mayor uso. Sin embargo, adicionalmente una interfaz por restricción de población se puede obtener limitando la población mediante la definición de una subclase de la clase servidora original (ver capítulo clases complejas) y definiendo una interfaz sobre dicha subclase.

Definición 33 ***Interfaz de proyección.*** *Siendo Cliente una especificación de cliente y Servidor una especificación de servidor, si* $\Sigma_{Servidor}$ *es la signatura de atributos y servicios de la clase del servidor determinado por Servidor, entonces una interfaz de proyección de Cliente respecto de Servidor está definida por una función* f *de la forma:*

$$f : (Cliente, Servidor) \rightarrow \Sigma'_{Servidor}$$

La función f define a qué atributos[1] y servicios tiene acceso el cliente restringiendo su percepción de la signatura del servidor[2], es decir, se cumple que $\Sigma'_{Servidor} \subseteq \Sigma_{Servidor}$. . Así, una interfaz de proyección, además de establecer un canal de comunicación, puede restringir los atributos y servicios accesibles por el cliente en la clase servidora.

Desde un punto de vista semántico, el mecanismo de interfaz puede verse como un tipo particular de herencia tal como se describe en [10]. Las tradicionales vistas del mundo de las bases de datos tienen cierta analogía con este concepto. Desde el punto de vista del comportamiento, los morfismos asociados a la justificación del mecanismo de interfaz deben expresar que sólo algunas propiedades se heredan en el objeto que resulta de la proyección.

4.5. Comunicación entre objetos

La formalización de OASIS está basada en la noción de ocurrencia de acciones en un cierto instante. Se asume por defecto que por cada

[1] Se asume por defecto que existe en el servidor (y para cada interfaz con él definida) un servicio que al ejecutarse genera una acción de respuesta hacia el cliente original. Dicha acción lleva un evento cuyos parámetros son los atributos que son accesibles por el cliente respecto del servidor. Esta interacción corresponde a una comunicación *call/return*, explicada más adelante.

[2] Este mecanismo de ocultación se justifica en el uso de morfismos, de forma análoga a como vimos en el capítulo de clases complejas.

acción a en la signatura de la clase, existe una acción $\neg a$ representando la no-ocurrencia de dicha acción.

En un instante determinado, una acción ocurre en un objeto si se provee un servicio o si se requiere un servicio de otro objeto (o a sí mismo). Del mismo modo, la no-ocurrencia de una acción en un objeto acontece cuando no se provee un servicio o cuando no se requiere un servicio en un cierto instante.

Una acción establece una comunicación entre un cliente y un servidor. Siendo a la acción $\langle Cliente, Servidor, Servicio\rangle$, cada vez que en el objeto $Cliente$ se activa una obligación asociada a la fórmula $\phi[\neg a]false$, la acción a debe acontecerle. Asociada a la ocurrencia de dicha acción en el cliente, en el objeto $Servidor$ ocurrirá a ó $\neg a$ según las fórmulas de prohibición definidas sobre a. Por otra parte, si en el servidor, para alguna de sus fórmulas $\neg\phi[a]false$ se cumple que $\models_w \neg\phi$, entonces ocurre $\neg a$, en caso contrario ($\models_w \phi$) ocurre a, siendo w el mundo en el que está el objeto. Nótese que la ocurrencia o no-ocurrencia de acciones (tanto en el cliente como en el servidor) depende de sus respectivos estados (mundos) en el instante de requerir y de proveer el servicio, respectivamente. El instante en el cual se provee el servicio es igual o posterior al instante en el que se requiere el servicio.

En OASIS, un objeto dispone de cuatro mecanismos de comunicación. Cada uno de ellos está caracterizado por los siguientes aspectos:

Síncrono vs. asíncrono. La comunicación será síncrona si el cliente de la acción a debe esperar hasta conocer si en el servidor ocurre a o $\neg a$. Un caso especial de comunicación síncrona se produce cuando el servidor es el que debe esperar a que en el cliente ocurra (se genere) una determinada acción (la solicitud de servicio hacia el servidor). Cuando la comunicación es síncrona existe una acción indicando la ocurrencia o no-ocurrencia de la acción esperada. La comunicación será asíncrona si no existe espera asociada.

Ocurrencia forzada vs. ocurrencia no forzada. La comunicación es de ocurrencia forzada para una acción a si su ocurrencia debe producirse tanto en el cliente como en el servidor, o en ninguno de ellos.

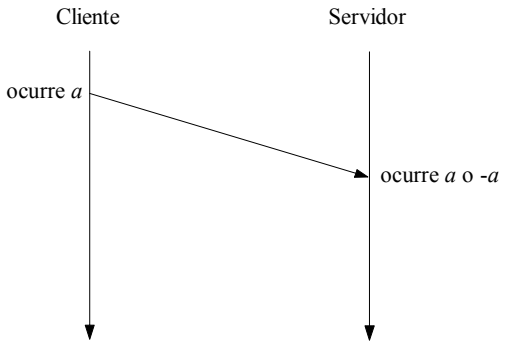

Figura 4.2: Representación del mecanismo *action sending*.

4.5.1. Mecanismos de comunicación

Los mecanismos de comunicación disponibles en OASIS son: action sending, action waiting, action calling y action sharing que se describen a continuación.

Action sending: asíncrono, ocurrencia no forzada. Este mecanismo se basa en la noción de obligación en el cliente. Sucede cuando es obligatoria la ocurrencia de una acción y el objeto en cuestión es el cliente de dicha acción. La Figura 4.2 ilustra este mecanismo[3].

Ejemplo 34 *En la especificación del objeto* `controlador(número,[1])` *se tiene la siguiente fórmula de obligación:*

```
nivel ⩾ max
        [¬valvula(nombre,[‘‘entrada’’])::cerrar] false
```

[3]En todas las figuras mostradas en este apartado las líneas verticales representan el paso del tiempo de arriba hacia abajo. Si la línea es continua indica que el objeto no está en espera. Si la línea es punteada indica que el objeto está en espera. Además, nos abstraeremos de aspectos asociados a cómo una acción es manipulada, tratada como un mensaje y transmitida desde el cliente hacia el servidor.

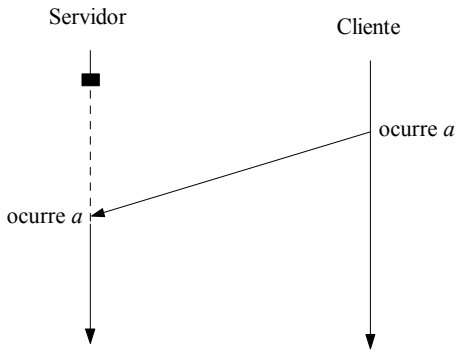

Figura 4.3: Representación del mecanismo *action waiting*.

Cuando se satisface `nivel`$\geqslant$`max`, al objeto cliente controlador debe acontecerle la acción:

```
<self,valvula(nombre,[‘‘entrada’’]),cerrar>
```

En un instante igual o posterior al actual, al objeto servidor `válvula(nombre,[‘‘entrada’’])`y de acuerdo con su estado, le acontecerá:

```
<controlador(número,[1]),
        valvula(nombre,[‘‘entrada’’]),cerrar>
```

o simplemente no ocurre dicha acción por no ser admitida.

Action waiting: síncrono, ocurrencia forzada. Este mecanismo también se asocia a la noción de obligación. A diferencia del *action sending*, el objeto en el cual se obliga a que ocurra la acción no es el cliente de la acción sino el servidor. Es decir, el servidor está obligado a esperar que ocurra la acción en el cliente. La Figura 4.3 ilustra este mecanismo.

Ejemplo 35 *En la especificación del objeto* `sensor(tipo,[‘‘nivel’’])` *se tiene la siguiente fórmula de obligación:*

```
conectado=false
        [¬controlador(someone):set(temperatura)] false
```

Cuando en dicho objeto se satisface `conectado=false` es obligatorio que le acontezca la acción:

```
<controlador(someone),
        sensor(tipo,[‘‘nivel’’]),set(temperatura)>
```

Por tratarse de una obligación sobre un servicio provisto, el objeto servidor sensor esperará hasta que ocurra la acción asociada.

Al implantar este tipo de interacción existiría la posibilidad de establecer un *timeout* por defecto, un tiempo máximo durante el cual la obligación puede afectar al servidor, es decir, cuánto tiempo esperará el servidor para recibir cierta solicitud de servicio desde un cliente. Una vez alcanzado el *timeout*, en el servidor ocurriría $\neg a$. Manipulando directamente el tiempo en la fórmula que define la obligación podría también especificarse un *timeout* arbitrario. Por ejemplo, redefiniendo el caso anterior con un *timeout* de 10 segundos:

```
(currenttime[4]-timeinicio) ⩽ 10 and conectado=false
        [¬controlador(someone):set(temperatura)] false
```

Siendo `timeinicio` un atributo de dominio *time* afectado por alguna evaluación asociada a una acción de obligación por satisfacerse `conectado=false`.

Del mismo modo, mediante las pertinentes evaluaciones y obligaciones podría representarse el hecho de alcanzar el timeout sin haberse cumplido la obligación.

Action calling: síncrono, ocurrencia forzada. Este mecanismo establece que una acción ocurre en el cliente y en el servidor o en ninguno de ellos[5]. La Figura 4.4 ilustra este mecanismo. En la imagen (a) la acción a ocurre en el cliente y en el servidor, en la imagen (b) la acción a no ocurre en ninguno de ellos.

[4] Asumiremos que `currenttime` es un atributo de dominio *time*, implícito para cada objeto y que representa la fecha y hora actual. Este atributo se ve afectado por cada *tick* del reloj.

[5] Corresponde a la *relación call* entre agregado y componentes, tratada en el capítulo de clases complejas.

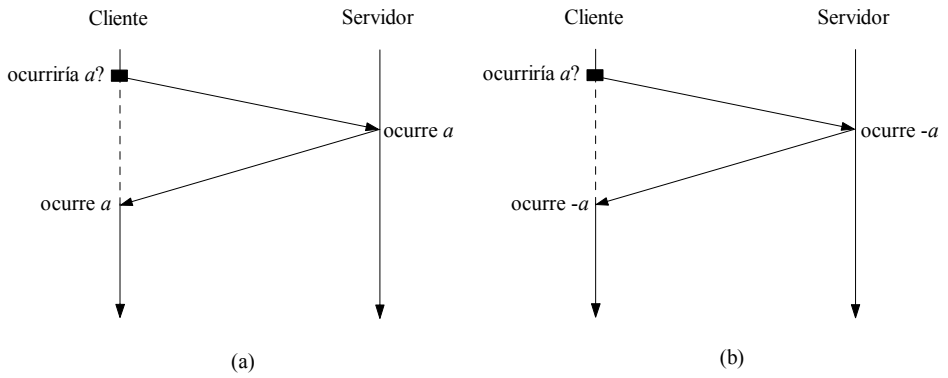

Figura 4.4: Representación del mecanismo *action calling*.

Ejemplo 36 *En la especificación de la clase* `coche` *(clase agregada) se tiene la siguiente interacción con su objeto componente de la clase* `motor`*:*

```
arrancar ⇒ motor.encender
```

Así, una acción con el servicio `arrancar` sólo ocurre si también ocurre la acción con el servicio `encender` en el objeto componente motor.

Action sharing: síncrono, ocurrencia forzada. Una interacción *action sharing* se produce cuando acciones en distintos objetos y asociadas a proveer un servicio deben ocurrir todas o ninguna de ellas[6]. Esta comunicación implica una sincronización entre objetos servidores.

Ejemplo 37 *En la especificación de la clase* `préstamo` *se tiene la siguiente interacción con sus componentes de la clase* `libro` *y de la clase* `socio`*:*

```
prestamo.prestar(Fecha,Socio,Libro) ⇔
        libro(codigo,[Libro]).ser_prestado(Fecha)
prestamo.prestar(Fecha,Socio,Libro) ⇔
        socio(numero,[Socio]).obtener_libro
```

Así, una acción con el servicio `prestar(Fecha,Socio,Libro)` sólo ocurre en el objeto `préstamo` si también ocurren las acciones con el servi-

[6]Corresponde a la *relación shared* entre agregado y componentes, tratada en el capítulo de clases complejas.

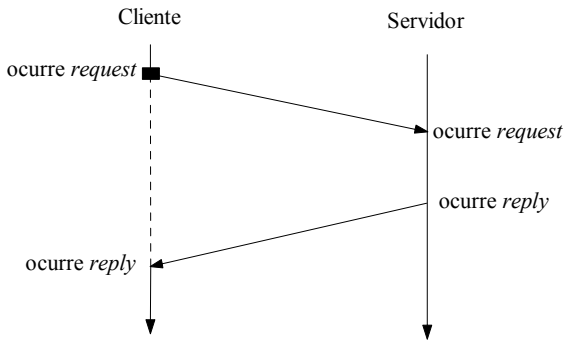

Figura 4.5: Representación de una interacción *call/return*

cio `ser_prestado(Fecha)` en el objeto componente `libro(código,[Libro])` y la acción `obtener_libro` en el objeto componente `socio(número,[Socio])`, y viceversa. Como la comunicación *action sharing* es de ocurrencia forzada, para que cualquiera de las tres acciones ocurra deben ocurrir también las otras dos restantes.

4.5.2. Comentarios adicionales

Mediante la combinación de *action sending* y *action waiting* es posible modelar cualquier interacción entre objetos pues ambas son primitivas elementales de comunicación. Sin embargo, para simplificar la especificación se proporcionan los mecanismos *action calling* y *action sharing* que establecen interacciones más complejas en un nivel adecuado de abstracción, resultando especialmente útiles para comunicación síncrona y de ocurrencia forzada entre un objeto agregado y sus objetos componentes.

Al especificar una particular interacción entre objetos, los mecanismos de comunicación disponibles son distintos dependiendo de qué clase se está especificando: la clase del cliente o la clase del servidor. Si se está especificando la clase del objeto cliente para una determinada acción, los mecanismos de comunicación disponibles son: *action sending* y *action calling*. Si se está especificando la clase del objeto servidor para dicha acción, los mecanismos de comunicación disponibles son: *action waiting* y *action sharing*.

Una interacción del tipo *request/reply* o *call/return* (*Remote Procedure Call* en arquitecturas distribuidas) puede ser especificada combinan-

do una comunicación *action sending* con una *action waiting*, tal como se ilustra en la Figura 4.5.

Capítulo 5

La metaclase OASIS

Este capítulo ha sido desarrollado por José Ángel Carsí Cubel.

La Ingeniería del Software se enfrenta a serios problemas que pueden determinar la viabilidad de las aplicaciones que soportan los sistemas de información. Existen numerosos estudios en los que se determina que uno de los factores críticos para el éxito de las aplicaciones consiste en su capacidad de evolución. Las razones que determinan la necesidad del cambio son muchas, entre ellas, la fundamental consiste en que los requisitos de los sistemas de información cambian con el tiempo. Así, toda aplicación debería evolucionar para no quedarse obsoleta.

A veces, es necesario mantener diferentes versiones de los sistemas para satisfacer las necesidades empresariales, bien porque se desea mantener información histórica, bien por la necesidad de disponer de diferentes versiones que reflejen diferentes aspectos de la realidad.

Por otra parte, las cada día más duras exigencias del mercado obligan a las empresas que se dedican a la construcción de software a reducir sus costes para ser competitivas. Uno de los paradigmas del software que se vislumbra como el más exitoso, de cara a esta reducción de costes, es aquél que reutiliza software ya desarrollado para construir nuevos sistemas.

Uno de los obstáculos con los que se enfrentan los analistas en su trabajo es la poca ayuda que se les proporciona en el sentido de pasos o guías a seguir en el proceso de construcción de los sistemas. Se necesitan guías metodológicas que asistan a los analistas en el proceso de desarrollo de los sistemas.

Un aporte significativo para solucionar la problemática planteada es la introducción del nivel meta. La vida de cualquier objeto comienza con

la ocurrencia del evento de creación de dicho objeto. Es difícil pensar que un objeto que no existe todavía pueda servir el evento que lo va a crear. Tendrá que ser otro objeto, que aporte dicho servicio, el que cree las instancias que poblarán el esquema. Con el objetivo de contemplar lo anterior, se amplía la definición de clase presentada en el capítulo 2.

Definición 34 ***Clase.*** *Una clase en OASIS está formada por:*

- Un **nombre**, una **plantilla** y una **función de identificación**.
- Una **factoría** de objetos que ofrece como servicios los de creación y destrucción de objetos.
- Un **almacén de objetos**: todo objeto creado pasa a formar parte del conjunto existencia[1] de la clase $ext_t(Clase)$ en un instante t.

Las clases son objetos con un estado y una dinámica capaces de servir acciones solicitadas por la sociedad de objetos. Dentro del marco OO que se ha definido, dichos objetos deben ser instancias de alguna clase. Dicha clase será una "clase de clases", también llamada *metaclase.*

Definición 35 ***Metaclase.*** *Una metaclase es una clase cuyas instancias, también llamadas* ***metaobjetos****, son objetos especiales que a su vez son clases de objetos. Todos los objetos son instancias de alguna clase y las clases son a su vez instancias de la metaclase. De esta manera, las clases son objetos de primera categoría y pueden ser tratadas de la misma manera que los objetos.*

Para cerrar el ciclo y no tener una jerarquía (potencialmente infinita) de metaclases se realiza el cierre reflexivo de la metaclase diciendo que es instancia de sí misma.

En la Figura 5.1 se puede ver la jerarquía que relaciona la metaclase, las clases y los objetos que van a poblar los esquemas conceptuales a través de la relación *is_instance_of.*

[1] También llamada *población.*

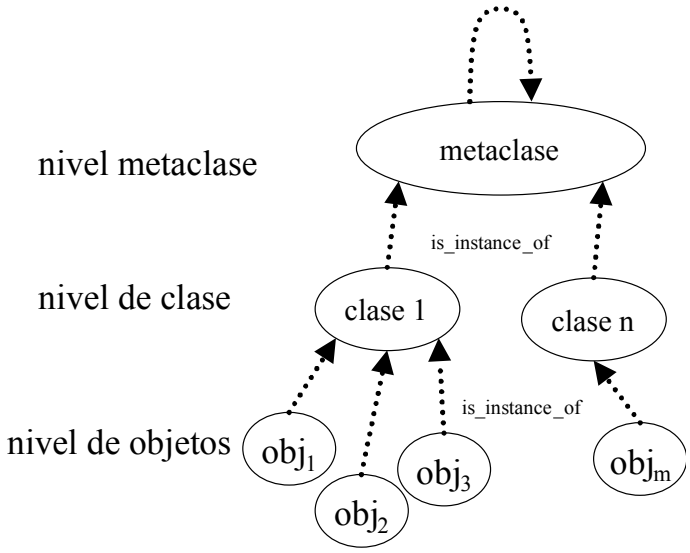

Figura 5.1: Jerarquía *is_ instance_ of.*

5.1. La doble visión de los metaobjetos

Las clases no sólo describen el estado y el comportamiento de los objetos a través de la plantilla, además pueden tener atributos y proporcionar servicios a nivel de clase gracias a su condición de objetos. Entre los servicios que proporcionan las clases se encuentran los eventos de creación y destrucción de objetos. Entre los atributos se encuentra el "siguiente_oid" y la población. Una clase puede ser vista como una agregación relacional en la que los componentes de la misma son sus instancias.

Las instancias de la metaclase pueden considerarse bajo una doble visión que se muestra en la Figura 5.2.

- Por un lado, son **clases** en las que se definen las propiedades que comparten todas las instancias. En la Figura 5.2 se puede observar que el metaobjeto cuyo oid es oid_X tiene en su parte izquierda el nombre, la definición de la plantilla (el conjunto de atributos, eventos, fórmulas, y procesos), la función de identificación, la factoría (eventos `alta_libro` y `baja_libro`) y el almacén (la población).

- Por otro lado, son **objetos** con un estado (representado por los atributos evaluados), y una dinámica (expresada por los ciclos de

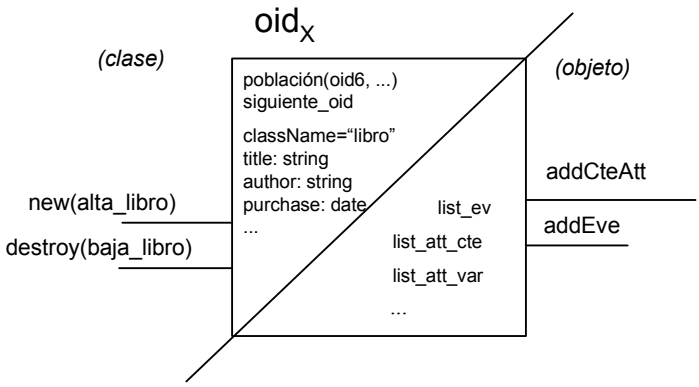

Figura 5.2: Ejemplo de la doble visión de un metaobjeto.

vida posibles). El estado de los metaobjetos almacena la plantilla de la clase en atributos variables que contienen todas las secciones que describen la misma. Así, los eventos que manipulan el estado son capaces de modificar la plantilla de la clase. En el ejemplo se pueden observar los servicios que proporciona el metaobjeto: añadir un atributo constante, añadir un evento, etc.

Llamaremos `class` a la clase que permite definir el comportamiento de las clases elementales y especificar los atributos, eventos, precondiciones, etc. En OASIS existen diferentes operadores de clases que permiten construir nuevas clases a partir de otras clases ya definidas. Como se sigue el criterio de dar las propiedades estructurales emergentes de cualquier clase compleja de la misma forma que las de una clase simple, los servicios que sirven para especificar las clases simples se necesitan, junto con otros (que añadan el resto de características de la clase compleja), para construir las clases complejas. Debido a esto, la metaclase se ve como una jerarquía de clases. La definición completa de las relaciones entre las clases que forman la metaclase OASIS se puede observar más adelante.

Las clases pueden desempeñar diferentes papeles en una especificación OASIS. Pueden ser componentes de una agregación (`component`). Pueden ser la clase compuesta de una agregación (`aggregated`). Pueden ser subclases de los diferentes tipos de relaciones de herencia que existen: particiones estáticas, dinámicas o grupos de rol (`static_sub`, `dynamic_sub`, `rol_sub`). Pueden ser la superclase de las particiones (`static_sup`, `dynamic_sup`, `rol_sup`).

Las particiones estáticas y dinámicas forman jerarquías de herencia (`hierarchy`) de forma que, del producto cruzado de las clases hoja de las jerarquías que tienen una superclase en común, aparecen las especies (`species`) que a su vez son clases.

$$\texttt{component} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{aggregated} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{component} \overset{part\_of}{\rightarrow} \texttt{aggregated}$$
$$\texttt{\{rol\_sub, dynamic\_sub, static\_sub\}} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{static\_sup} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{static\_sub} \overset{part\_of}{\rightarrow} \texttt{staticSpe\_sup}$$
$$\texttt{dynamic\_sup} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{dynamic\_sub} \overset{part\_of}{\rightarrow} \texttt{dynamic\_sup}$$
$$\texttt{rol\_sup} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{rol\_sub} \overset{part\_of}{\rightarrow} \texttt{rol\_sup}$$
$$\texttt{\{dynamic\_sup, static\_sup\}} \overset{rol\_of}{\rightarrow} \texttt{hierarchy}$$
$$\texttt{species} \overset{rol\_of}{\rightarrow} \texttt{class}$$
$$\texttt{\{static\_sub, dynamic\_sub, hierarchy\}} \overset{part\_of}{\rightarrow} \texttt{species}$$

Las relaciones $\overset{part\_of}{\rightarrow}$ anteriores representan agregaciones relacionales entre las clases. Como una subclase de una partición no puede ser subclase de otra partición, el grupo de rol de las subclase tiene de cardinalidad (0, 1). De la misma manera, una clase no puede ser la compuesta de dos agregaciones distintas por lo que la cardinalidad del rol de la clase `aggregated` es (0, 1). Existe una restricción adicional para las especies: la raíz de la jerarquía de herencia de las diferentes jerarquías por las que aparece la especie debe de ser única.

La metaclase puede ser vista como un editor abstracto de clases OASIS en el que la especificación del esquema del sistema se construye haciendo uso de los servicios que la metaclase proporciona obteniendo como resultado un conjunto de clases y sus relaciones. La población de la metaclase coincide, por lo tanto, con lo que en Ingeniería del Software se conoce como *repositorio*, es decir, la colección de elementos software de que dispone una herramienta para la construcción de nuevos sistemas por reutilización de componentes, así como las herramientas que permiten su recuperación y uso.

5.2. La IS desde la visión de la metaclase

En los siguientes puntos se presentan algunos de los problemas de la Ingeniería del Software (IS) y la vía de solución dentro del modelo OASIS presentado.

5.2.1. Evolución del software

Un esquema conceptual en OASIS, es el estado de un determinado objeto (la metainstancia que lo define). El estado de cualquier objeto puede ser modificado en cualquier momento por el usuario a través de los servicios ofrecidos por el metaobjeto si el usuario está autorizado a ello (capacidad capturada por el sistema a través del mecanismo de interfaces en OASIS). Estos cambios pueden seguir una aproximación temporal (marcando el tiempo en cada uno de los estados del objeto) o una dinámica (sólo se guarda el último estado). El efecto resultante de la ocurrencia de los eventos en el estado de los objetos se da de forma declarativa en la especificación de la metaclase. Una parte del estado de los metaobjetos es la plantilla que evoluciona. La otra parte la forma la población de la clase que deberá migrar para ser conforme con la primera. Ambos cambios se describen de la misma forma en la especificación de la metaclase.

5.2.2. Versionado del software

La evolución del software está motivada por la evolución de las reglas de negocio en el espacio del problema: "lo que es válido hoy no lo será mañana". Si se desea mantener una historia de los diferentes esquemas conceptuales, cada uno de ellos válido durante un periodo de tiempo, es necesario marcar el tiempo en los estados de los metaobjetos que codifican los esquemas conceptuales.

Cada estado de los metaobjetos está marcado por los periodos de tiempo en los que dicho estado permanece. En este sentido, el versionado de software es tratado en OASIS usando modelos temporales en el nivel meta.

5.2.3. Reuso del software

El software desarrollado usando OASIS facilita su reutilización. Los *assets*[2] son las clases y los objetos. El repositorio formado por la extensión de la metaclase es la biblioteca de componentes que sirve para producir software por reuso.

La biblioteca de componentes tiene una estructura (relaciones de herencia y agregación) que permite la optimización de las búsquedas realizadas por las herramientas para recuperar los *assets* a diferentes niveles de especialización (relación de herencia) o granularidad (relación de agregación).

El lenguaje de consulta usado para recuperar los componentes es el mismo que el usado para recuperar los objetos en las aplicaciones de los usuarios.

5.2.4. Ayudas y guías metodológicas

En el área de las herramientas CASE, las metodologías subyacentes proporcionan lenguajes visuales y un tedioso conjunto de reglas que el desarrollador debe seguir para describir los modelos conceptuales.

Ésta es una de las razones del poco éxito de las herramientas CASE tradicionales. En la práctica, el desarrollador produce software basándose parcialmente en el método que le ofrecen las herramientas CASE y aplicándolo según su criterio.

Es obvio, que el único camino para que un método sea aplicado es implementarlo como un conjunto de ayudas y herramientas sobre el ordenador.

En OASIS, el protocolo de la metaclase puede determinar de forma clara y no ambigua la forma en la que se debe construir software. La metodología también está codificada en el modelo OASIS y las mismas herramientas que se usan para validar las aplicaciones sirven como ayuda y guía automática de la metodología.

[2]Los componetes reutilizables.

Parte II

OASIS como lenguaje

Capítulo 6

Especificación del sistema

En OASIS, el modelo conceptual se corresponde con un conjunto de elementos de especificación, incluyendo dominios, clases simples, clases complejas e interfaces. En este capítulo cada apartado aborda un elemento de la especificación con su sintáxis y ejemplos. En el capítulo siguiente se presenta un ejemplo más extenso de un sistema de información modelado usando OASIS.

Para precisar la sintaxis del lenguaje, junto a la descripción de cada ítem se presentará la porción de BNF asociada. La notación utilizada es:

- **símbolo terminal** o si se trata de caracteres, entre ' '
- <**símbolo no-terminal**>
- símbolos opcionales, están entre [ ]
- símbolo alternativos, están separados por |

Además, al exponer la sintaxis de un determinado elemento del lenguaje se usarán las siguientes simplificaciones: siendo x un elemento en particular, para describir un bloque de elementos x usaremos <bloque_x>; para una lista de elementos x usaremos <lista_x> y para una secuencia de elementos x usaremos <sec_x>. Los significados asociados a cada uno de ellos se establecen a continuación.

Sintaxis

<bloque_x>	::=	<bloque_x> <x> | <x>
<lista_x>	::=	<lista_x> ',' <x> | <x>
<sec_x>	::=	<sec_x> <x> ';'| <x> ';'

En el anexo B se proporciona la sintaxis completa de OASIS 3.0. Para no hacer demasiado extensa la descripción de cada elemento de la especificación, algunos no terminales comunes sólo se detallan en el anexo B. A continuación se explican algunos símbolos no terminales usados frecuentemente.

- El no terminal <fórmula> hace referencia a una fórmula bien formada en lógica de predicados de primer orden, basada en valores constantes, atributos y/o parámetros de acciones.
- El no terminal <expresión> es una expresión *boolean* o aritmética basada en valores constantes, atributos y/o parámetros de acciones.
- El no terminal <valor> es un elemento de algún *sort* de los dominios permitidos.

En cada uno de ellos, los valores, atributos y/o parámetros involucrados deben ser consistentes respecto del contexto de especificación en el cual se están utilizando.

El modelo conceptual de un sistema de información se orresponde en OASIS con la especificación de un esquema conceptual.

Sintaxis

<def_esquema_conceptual>	::=	**conceptual schema** <id_esquema> [<bloque_def_dominio>] <bloque_def_clase> [<bloque_def_clase_compleja>] <bloque_def_interfaz> **end conceptual schema**

A continuación veremos en detalle la especificación de cada uno de los bloques mencionados.

6.1. Tipos de datos (dominios)

En primer lugar, veremos cómo definir los tipos de datos básicos. Estos se utilizarán como dominios para los valores de variables y atributos, y constituyen el nivel de especificación de datos básicos sobre el cual se definirán las clases del sistema (simples y complejas).

En OASIS hay varios tipos de datos estándar predefinidos[1]: **nat**, **int**, **real**, **bool**, **char**, **string**, **date**, **time** y **blob** ("binary large objects"). Sin embargo, el diseñador tiene la posibilidad de declarar otros nuevos tipos abstractos de datos y utilizarlos.

Sintaxis

<def_dominio>	::=	**domain** <id_dominio> '**(**' <lista_id_dominio> '**)**'
		[**import** <lista_id_dominio>] '**;**'
		[**var** <lista_def_variable>] '**;**'
		functions <sec_def_función>
		equations <sec_def_ecuación>
		end domain
<def_variable>	::=	<id_variable> '**:**' <dominio>
<def_función>	::=	<id_función> ['**(**' lista_def_parámetro '**)**']
		'**:**' <dominio>
<def_ecuación>	::=	<fórmula_ecuacional>

Observaciones

- La sección de importación nos permite utilizar tipos abstractos de datos previamente definidos. Estos son especificaciones algebraicas estándar de tipos abstractos de datos.

[1] Asumiremos disponibles las funciones típicas para números, cadenas de caractéres y tiempo. Puede ser conveniente utilizar directamente las funciones del lenguaje de prototipación o generación de código. Por defecto utilizaremos las definidas en la biblioteca estándar ANSI C, declaradas dentro de los siguientes *headers*: `<math.h>` para **nat**, **int** y **real**, `<ctype.h>` y `<string.h>` para **char** y **string**, y `<time.h>` para **time**. Además, **date** es un subsort de **time**, incluyendo sólo año, mes y día.

- Los tipos de datos genéricos pueden especificarse colocando una lista de dominios entre paréntesis, indicando que tales tipos de datos están parametrizados por otros tipos base. De esta forma, el tipo de datos definido es una plantilla para otros tipos, los cuales se pueden obtener instanciando los identificadores con clases primitivas o dominios previamente definidos.

- La sección **equations** es una secuencia de fórmulas bien formadas de la lógica ecuacional, del tipo $ter_1 = ter_2$, donde ter_1 y ter_2 son términos que describen cómo trabajan las funciones.

Ejemplo 38 *La especificación de una* `pila` *es un dominio genérico, por esto está parametrizado por el tipo de ítem que contiene la pila. En el cuerpo de cualquier especificación de una clase del sistema, puede haber una instancia de pilas de números naturales:* `pila(nat)`*, pila de contratos:* `pila(contratos)`*, etc.*

```
domain pila(item)
var P:pila,N:item
functions
    new:pila;
    push(P,N):pila;
    pop(P):pila;
    top(P):item;
equations
    pop(push(P,N))=P;
    top(push(P,N))=N;
end domain
```

6.2. Especificación de clase

Una clase en su definición consta de un nombre de clase y una plantilla. La plantilla de clase está dividida en secciones o párrafos.

Sintaxis

<def_clase>	::=	**class** <id_clase>
		<mecanismos_identificación>
		<atributos_constantes>
		[<atributos_variables>]
		[<atributos_derivados>]
		[<derivaciones>]
		[<restricciones_integridad>]
		<eventos>
		[<operaciones>]
		[<disparos>]
		[<evaluaciones>]
		[<precondiciones>]
		[<protocolos>]
		end class

Observaciones

- Además de las opcionalidades explícitamente mostradas, si una clase está definida por herencia a partir de otra clase entonces todas las secciones son opcionales.

- Si <id_clase> coincide con el identificador del esquema conceptual se asume que se trata de la descripción de propiedades globales asociadas a la clase implícitamente establecida por agregación de todas las clases definidas en el esquema conceptual.

A continuación se describe en detalle cada una de las secciones de la plantilla de una clase.

6.2.1. Mecanismos de identificación

La sección de identificación de una especificación OASIS define los mecanismos de identificación utilizados para referirse a los objetos. Un mecanismo de identificación es una función que establece correspondencias entre valores de atributos de un objeto y su *oid*. De esta forma, un mecanismo de identificación es una clave para hacer referencia a objetos en el espacio del problema.

Sintaxis

<mecanismos_identificación>	::=	**identification** <sec_def_identificador>
<def_identificador>	::=	<id_identificador> ':' '(' <lista_id_atributo> ')'

Observaciones

- Se pueden definir distintos mecanismos de identificación para una misma clase. Un mecanismo de identificación está compuesto por una lista de atributos constantes. La lista de valores asociada a dichos atributos constantes debe determinar unívocamente un objeto de la clase.

Ejemplo 39 *En la clase* `cuenta`*, el mecanismo de identificación* `por_número`*, basado en el atributo constante* `número`*.*

```
por_numero:(numero);
```

Ejemplo 40 *Los objetos de la clase* `receta_cocina` *se identifican por un* `código_receta` *o alternativamente por el* `nombre_plato` *junto al* `nombre_creador`*.*

```
codigo:(codigo);
nombre:(nombre_plato,nombre_creador);
```

6.2.2. Atributos

Como se ha mencionado, un objeto se puede ver como un proceso observable. El estado de los objetos se observa mediante los valores de sus 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*.

Atributos constantes. Aquellos que toman su valor cuando se crea el objeto y no lo cambian a lo largo de toda su vida.

Atributos variables. Aquellos cuyos valores pueden cambiar cuando ocurre una acción relevante.

Atributos derivados. Aquellos cuyos valores se dan en términos de los valores de otros atributos. Los atributos derivados serán, a su vez, constantes o variables según la naturaleza de los atributos que los definan.

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. Por su similaridad con la signatura implícita para componentes en una agregación, el detalle de dichas funciones y operaciones se presenta en la sección dedicada a la agregación.

Atributos constantes y variables

Las declaraciones de atributos constantes y variables utilizan una misma sintaxis, sin embargo, son agrupados en secciones distintas.

Sintaxis

<atributos_constantes>	::=	**constant attributes** <sec_def_atributo_cons_o_var>
<atributos_variables>	::=	**variable attributes** <sec_def_atributo_cons_o_var>
<def_atributo_cons_o_var>	::=	<id_atributo>':' <dominio> ['**(**'<lista_valor_por_defecto>'**)**'] [**towards** <cardinalidad> [<def_representación>]]
<valor_por_defecto>	::=	<valor>
<cardinalidad>	::=	'**(**' <card_min> '**,**' <card_max> '**)**'
<def_representación>	::=	**list** '**[**' <def_índice> '**]**' | **set** | **bag**
<def_índice>	::=	<rango> | <lista_cadena>

Observaciones

- Los atributos pueden tener valor por defecto cuando se crea el objeto. El diseñador puede introducir entre paréntesis este valor por defecto en la declaración del atributo.
- **towards** <cardinalidad> presenta las cardinalidades vistas desde el objeto hacia el dominio primitivo del atributo. Si no se especifica se asume **towards**(0,1).
- Debe cumplirse que <card_min> $\leqslant$ <card_max>.
- Los valores de atributos constantes y variables son parámetros implícitos del evento **new** del objeto. Si para algún atributo se tiene que <card_min> $\geqslant 1$, entonces es obligatorio que a partir de la creación del objeto dicho atributo tenga al menos esa cantidad de valores. Si el parámetro asociado a un atributo constante o variable no está instanciado en el evento **new** y existe un valor por defecto declarado, entonces el atributo tomará dicho valor.
- Cuando el atributo es multivaluado (<card_max> > 1), **list**, **set** y **bag** permiten indicar cuál es la representación escogida. En **list** y **bag** se permiten elementos duplicados, en **set** no. En **list** la declaración <def_índice> permite definir la forma de referencia usada para acceder a los elementos de la lista. Si no se especifica **list**, **set** y **bag**, se asume que es de tipo **set**.

Ejemplo 41 *Una declaración del atributo* números_teléfono *de dominio* **string**, *multivaluado y con representación* **list**, *sería:*

```
numeros_telefono:string towards(0,*) list[1..*];
```

El '*' indica que no existe restricción respecto de la cardinalidad máxima (en **towards**), y en el máximo valor del rango (en **list**).

Ejemplo 42 *El mismo ejemplo, pero estableciendo una inicialización de tres números por defecto sería:*

```
numeros_telefono:string(``616381'',``647781'',``505753'')
        towards(0,*) list[1..*];
```

Atributos Derivados

En OASIS es posible declarar atributos que se definen en términos de otros. La sección **derived attributes** se usa para tal fin.

Sintaxis

<atributos_derivados>	::=	**derived attributes** <sec_dec_atributo_derivado>
<dec_atributo_derivado>	::=	<id_atributo> ':' <dominio>

Observaciones

- Para atributos derivados, el dominio declarado debe ser consistente con su definición en la sección **derivations**, la cual a su vez determina si se trata de un atributo constante o variable, y su cardinalidad y representación.

6.2.3. Derivaciones

La sección **derivations** se usa para especificar la definición de un atributo derivado.

Sintaxis

<derivaciones>	::=	**derivations** <sec_def_atributo_derivado>
<def_atributo_derivado>	::=	['**{**'<fórmula>'**}**'] <asignación>
<asignación>	::=	<id_atributo> '**:=**' <expresión>

Ejemplo 43 *Una derivación en la clase* `cuenta` *define que el atributo derivado* `en_números_rojos` *(y de dominio* **bool***) tiene el valor* `true` *cuando el saldo de la cuenta es menor que cero.*

```
{saldo >= 0} en_numeros_rojos := false;
{saldo < 0} en_numeros_rojos := true;
```

o de modo más breve, pero equivalente:

```
en_numeros_rojos := {saldo < 0};
```

Ejemplo 44 *El* `crédito` *que dispone el cliente corresponde al* `saldo` *de la cuenta más 50.000 pesetas.*

```
credito := saldo + 50000;
```

6.2.4. 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.

Sintaxis

```
<restricciones_integridad>  ::=  constraints <sec_def_restricción>
<def_restricción> ::=
                  [ always ] '{'<fórmula>'}'
                  | sometimes '{'<fórmula>'}'
                  | next '{'<fórmula>'}'
                  | <fórmula_después> since <fórmula_antes>
                  | <fórmula_antes> until <fórmula_después>
                  | sometimes ['('<timeout>')'] <fórmula_después>
                                since <fórmula_antes>
                  | always ['('<delay>')' ] <fórmula_después>
                                since <fórmula_antes>
<fórmula_antes>          ::=  '{'<fórmula>'}'
<fórmula_después>        ::=  '{'<fórmula>'}'
<timeout>                ::=  <op_rel_timeout> <natural>
                                    <unidad_tiempo>
<delay>                  ::=  <op_rel_delay> <natural>
                                    <unidad_tiempo>
<op_rel_timeout>         ::=  '<' | '<='
<op_rel_delay>           ::=  '>' | '>='
<unidad_tiempo>          ::=  seconds | minutes| hours | days
                              | weeks | months | years
```

Observaciones

- Si al ser verificadas las restricciones de integridad alguna de ellas no se satisface, el siguiente estado del objeto es el último en el que se cumplieron las restricciones de integridad. Es decir, los cambios de estado posteriores a la última situación de integridad son ignorados.
- Una restricción de integridad que utiliza el operador **sometimes** sin especificar un límite de tiempo, tiene un sentido equivalente al de una precondición para el evento **destroy** del objeto.

Ejemplo 45 *Los siguientes son ejemplos de restricciones de integridad:*

```
{interes <= 15};

{valvula=‘‘abierta’’} until {nivel > 100};

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

```
always (> 3 months){proyecto_terminado=true}
            since {proyecto_inscrito=true};
```

6.2.5. Eventos

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

Sintaxis

<eventos>	::=	**events**
		<sec_def_evento>
<def_evento>	::=	
		<id_evento> ['**(**' lista_def_parámetro '**)**']
		[**new** | **destroy**]
		[**alias for** <operación_implícita> | <coordinación>]
<coordinación>	::=	**calling to members**
		<lista_servicio_componente>
		| **sharing with members**
		<lista_servicio_componente>

Observaciones

- **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 implícitamente como parámetros los valores para los atributos constantes y variables del objeto. Si se trata de la especificación de una clase simple estos eventos deben ser explícitamente declarados.

- El evento **new** crea al objeto y debe ser forzosamente el primer evento en la vida del objeto (aunque no necesariamente el único evento en dicho primer paso, como puede ocurrir en la creación de objetos agregados).

- Existen peculiaridades asociadas al evento **new** relativas a la creación de objetos complejos y que serán descritas con detalle más adelante.

- El evento **destroy** es la causa de la destrucción de un objeto. Si una clase dada no tiene declarado dicho evento asumiremos que los objetos de la clase en cuestión son "inmortales".

- Usando <operación_implícita> es posible establecer un alias para alguna operación implícita, las cuales se detallan más adelante en el apartado de agregación.

- 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. Los eventos compartidos pueden utilizarse cuando los componentes de un agregado son de tipo relacional (ver apartado de clases complejas).

- Los eventos implicados y los compartidos sólo se definen en una clase agregada. En ambos casos, debe establecerse una correspondencia entre el evento del agregado y el servicio del componente, mediante la definición de <lista_servicio_componente>.

Ejemplo 46 *Para una clase* `transferencia` *(una agregación de dos objetos de la clase* `cuenta`*), el evento* `transferir` *implica la ocurrencia de los eventos* `reintegro` *y* `depósito` *en sus componentes. La ocurrencia de dichos eventos se produce en los tres objetos o en ninguno de ellos*[2]*. En este caso es necesario definir eventos implicados (en un sólo sentido) para permitir que reintegros o depósitos efectuados directamente sobre una cuenta no impliquen un evento* `transferir` *en un objeto de la clase* `transferencia`*.*

[2] Al describir la sección de operaciones se mostrará una forma alternativa de especificar esta situación mediante el uso del concepto de *transacción*.

```
class transferencia
     ...
events
    transferir[3](Desde:nat,Hacia:nat,Cantidad:int) new
        calling to members
            cuenta(numero,[Desde]).reintegro(Cantidad),
            cuenta(numero,[Hacia]).deposito(Cantidad);
    ...
end class


class cuenta
    ...
events
    reintegro(Cantidad:int);
    deposito(Cantidad:int);
    ...
end class
```

Ejemplo 47 *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. 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

class libro
    ...
events
    ser_prestado(Fecha:date);
    ...
end class
```

[3]Nótese que este evento corresponde al evento **new** en la clase transferencia.

```
class socio
    ...
events
    obtener_libro;
    ...
end class
```

6.2.6. 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 anterior al de ocurrencia de la acción y ϕ se satisface en el instante inmediatamente posterior a la ocurrencia de a. Pueden especificarse diferentes evaluaciones asociadas a una misma acción pero con distintos ψ y/o ϕ.

Sintaxis

<evaluaciones>	**::=**	**valuations** <sec_def_evaluación>
<def_evaluación>	**::=**	['**{**'<fórmula>'**}**'] '**[**' <acción> '**]**' <lista_asignación>
<asignación>	**::=**	<id_atributo> '**:=**' <expresión>

Observaciones

- En lugar de utilizar un "**=**" en la fórmula que caracteriza el estado alcanzado se usa "**:=**" por ser más intuitiva su interpretación. Del mismo modo, en lugar de utilizar **and** para conectar cada aserción (con el sentido de asignación) con respecto al valor de los atributos, simplemente se separan con una "**,**".

- Los atributos constantes y derivados no deben ser un atributo de asignación. Por otro lado, todo atributo variable debería tener asociada al menos una asignación en alguna evaluación.

- El dominio resultante de la <expresión> debe ser consistente con el dominio del atributo.

Ejemplo 48 *El atributo* `bloqueada` *(de una cuenta bancaria) establece si una* `cuenta` *puede o no registrar reintegros. Este atributo se ve afectado por el evento* `bloquear`*.*

```
{bloquedada=false}[bloquear] bloqueada:=true;
```

Si no se declara una condición de evaluación (ψ) se asume *true*, es decir, la evaluación es aplicable en cualquier estado del objeto. En el ejemplo anterior, si el bloqueo se produce sin importar si la cuenta está bloqueada o no, entonces se puede simplificar a:

```
[bloquear] bloqueada:=true;
```

Ejemplo 49 *Una evaluación que establezca una cierta relación entre* `depósito(Cantidad:nat)` *y* `reintegro(Cantidad:nat)` *respecto del atributo* `saldo` *de la cuenta.*

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

Este tipo de evaluaciones, se puede simplificar para hacerla más intuitiva y cercana al concepto de asignación de valores para los atributos. Así, el caso anterior puede reescribirse como:

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

Esta simplificación es sólo "azúcar sintáctico" puesto que en la fórmula `saldo:=saldo+Cantidad`, la primera aparición de `saldo` representa a `saldo` en el estado alcanzado y la segunda aparición corresponde a `saldo` en el estado de partida.

6.2.7. 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 el 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 $\neg a$.

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".

Sintaxis

<precondiciones>	**::=**	**preconditions** <sec_def_precondición>
<def_precondición>	**::=**	<acción> **if** '**{**'<fórmula>'**}**'

Observaciones

- Los eventos de creación y destrucción (**new** y **destroy**) tienen como precondición por defecto la no-existencia y existencia de la instancia que se va a crear o destruir, respectivamente. Adicionalmente, es posible incluir en la propia especificación de la clase precondiciones a dichos eventos.

Ejemplo 50 *El evento* `reintegro(Cantidad:nat)` *en la clase* `cuenta` *tiene la siguiente precondición:*

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

Así, la cuenta no realizará un reintegro solicitado por el usuario si no se tiene un saldo mayor o igual a la cantidad especificada.

6.2.8. 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.

Para caracterizar la actividad declarada en el párrafo de disparos del lenguaje OASIS utilizaremos 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".

Sintaxis

<disparos>	::=	**triggers**
		<sec_def_disparo>
<def_disparo>	::=	<acción> **when** '**{**'<fórmula>'**}**'

Observaciones

- El objeto que tiene la definición del disparo debe ser cliente, servidor o ambos en la acción.
- Los disparos, por representar una obligación, deben ser consistentes con los permisos definidos mediante precondiciones y protocolos. Es decir, cuando una obligación se hace efectiva, debe estar permitida.
- La obligación establecida por un disparo se mantiene mientras la condición de disparo se satisface, es decir, el objeto continuará solicitando un determinado servicio o esperando[4] por ofrecer cierto servicio, según corresponda.

Ejemplo 51 *En la clase* `cuenta` *es posible incluir un disparo para expresar que en el momento en que el* `crédito` *sea mayor que* `límite` *se debe fijar un nuevo tipo de interés en un diez por ciento (mediante el evento* `fijar_interés(Interés:nat))`*:*

```
self::fijar_interes(10)
            when {credito>limite and tipo_interes<>10}
```

La condición del disparo incluye comprobar el valor del tipo de interés. Si no se incluyera esto, y considerando la definición del disparo de OASIS, el objeto generará la acción del disparo mientras la condición se satisfaga.

[4]Esto corresponde al concepto de *action waiting* tratado en el capítulo de interacción entre objetos.

Ejemplo 52 *Se generan mensajes de advertencia desde un objeto de la clase* ***impresora*** *(identificado por el atributo* ***dispositivo****) hacia el* ***controlador(clave,[‘‘C1’’])*** *encargado de las incidencias de las impresoras.*

```
controlador(clave,[‘‘C1’’])::
    incidencia(dispositivo,fecha,hora,‘‘tinta agotada’’)
        when {porcentaje_tinta < 10 and avisado_tinta=false};

controlador(clave,[‘‘C1’’])::
    incidencia(dispositivo,fecha,hora,‘‘atasco papel’’)
        when {atasco=true < 10 and avisado_atasco=false};
```

Ejemplo 53 *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. Al llegar a un 95 % de utilización se les avisa a los usuarios conectados que el sistema puede hacerse inestable.*

```
usuario(everyone)::aviso(fecha,hora,‘‘libere espacio disco’’)
    when {porcentaje_utilizado > 80
            and avisado_80_porciento=false};

usuario_conectado(everyone)::
    aviso(fecha,hora,‘‘posible caida sistema’’)
        when {porcentaje_utilizado > 95
                and avisado_95_porciento=false};
```

Ejemplo 54 *En un cajero automático, un disparo que obligue al objeto a esperar (action wainting) una acción llamada* ***set*** *solicitada por algún objeto de la clase* ***administrador****, cada vez que se satisface* ***vacio=true****.*

```
administrador(someone):set when {vacio=true}
```

6.2.9. Operaciones y protocolos

Existen dos secciones en las cuales se describen especificaciones de proceso: *operations* y *protocols*. Aunque la sintaxis de ambas secciones es similar, la sección de operaciones especifica secuencias de acciones que obligatoriamente deben ocurrir, en cambio, la sección de protocolos especifica secuencias de acciones cuya ocurrencia está permitida (pueden ocurrir).

Sintaxis

<operaciones>	::=	**operations** <bloque_def_operación>
<def_operación>	::=	<id_operación>['**(**'lista_def_parámetro'**)**'] [**transaction**] '**:**'<sec_def_estado>
<protocolos>	::=	**protocols** <bloque_def_protocolo>
<def_protocolo>	::=	<id_protocolo> ['**(**'lista_def_parámetro'**)**'] '**:**' <sec_def_estado>
<def_estado>	::=	<id_estado>'**=**'<términos_estado>
<términos_estado>	::=	<un_término_estado> | <términos_estado> '**+**' <términos_estado>
<un_término_estado>	::=	<término_con_guarda> | <término_sin_guarda>
<término_con_guarda>	::=	'**{**'<fórmula>'**}**'<término_sin_guarda>
<término_sin_guarda>	::=	<acción_proceso> ['**.**' <id_estado>] | <proceso> ['**.**' <id_estado>]
<acción_proceso>	::=	[<cliente_proceso>] [<servidor_proceso>] <servicio>
<cliente_proceso>	::=	<cliente> | **client** '**:**'
<servidor_proceso>	::=	<servidor> | **server** '**::**'
<proceso>	::=	<id_operación> ['**(**' lista_parámetro '**)**'] | <id_protocolo> ['**(**' lista_parámetro '**)**']

Observaciones

- Para cada operación o protocolo debe existir un <id_estado> que coincida con el <id_operación> ó <id_protocolo>, respectivamente. Este <id_estado> representa el estado de inicio del proceso.
- El estado de proceso representado por **0** no se especifica. Cuando <término_sin_guarda> se reduce sólo a <acción_proceso> o <proceso> (sin especificar un estado de proceso siguiente), se asume que el siguiente estado de proceso es **0** (fin del proceso).
- La palabra reservada **transaction** indica que la operación actúa como transacción. Así, una transacción es un caso particular de

operación. Durante la ejecución, en el contexto de una transacción, todas las operaciones anidadas (si existen) son tratadas también como transacciones.

Uso de parámetros en procesos

Tal como los eventos, los procesos pueden tener parámetros. Los subprocesos anidados, involucrados en acciones utilizadas en la especificación de un proceso, también pueden tener parámetros. Los tipos de los parámetros deben ser coherentes con la declaración del evento o del proceso, según corresponda.

Los parámetros de un proceso pueden ser utilizados en guardas (<fórmula>) o en expresiones que definen otros parámetros de eventos o subprocesos.

El cliente y el servidor de una acción son, por defecto, parámetros de toda acción. En la especificación de un proceso para referirse al cliente y al servidor de la acción que invoca al proceso se utilizan los identificadores de parámetros **client** y **server**, respectivamente.

Al usar los eventos y procesos en especificaciones de proceso (o en el resto de secciones de la plantilla) un parámetro puede estar determinado por una expresión. El número y tipo de los parámetros (*sort* del resultado de la expresión) deben ser consistentes con lo especificado en la declaración correspondiente. Se asume que los parámetros de acciones que solicitan un servicio se instancian totalmente de acuerdo con las expresiones que los especifican.

Tratamiento de comunicación call/return

Una acción *call/return* es un tipo particular de acción que modela una comunicación sincrónica entre un cliente y un servidor. El cliente genera una acción solicitando un servicio del servidor y espera a que le acontezca una acción que representa la respuesta del servidor al servicio solicitado.

Esta comunicación es ampliamente usada en los lenguajes de programación en forma de invocaciones a procedimientos. En lenguajes secuenciales, una invocación incluye: el traspaso del hilo de ejecución al procedimiento invocado y la transferencia de información hacia y desde el procedimiento invocado. En el contexto de OASIS, quien invoca es el

objeto cliente y quien es invocado es el objeto servidor. Además, en OASIS se asume por defecto que todos los objetos actúan concurrentemente por lo cual el cliente no cede su hilo de control. Esto hace necesario contar con un mecanismo de espera en el cliente y obtener así el efecto deseado de una interacción *call/return*. Desde el punto de vista de las acciones acontecidas, la llamada y el retorno son dos acciones distintas y ambas deben ocurrir tanto en el objeto cliente como en el objeto servidor.

Por simplicidad, en la especificación de una operación en el cliente se especificará sólo con una acción. Esta acción tendrá parámetros de entrada y/o de salida. Un ';' marca la separación entre la lista de parámetros de entrada (puestos a la izquierda) y de salida (puestos a la derecha). Implícitamente, esta facilidad sintáctica se interpreta como la secuencia obligada de dos acciones, como si se tratara de una operación. La primera corresponde a una solicitud de servicio con los parámetros de entrada y hacia un determinado servidor (parte *call*). La segunda es un servicio requerido generado por el servidor de la acción (parte *return*) con los parámetros de salida.

Analicemos la siguiente situación como ejemplo. Sea `c` la referencia del objeto cliente, `s` la referencia del objeto servidor y `op(10,5;x)` el servicio solicitado por el objeto `c` al objeto `s`. El servicio realizará la multiplicación de los argumentos de entrada, retornando el parámetro de salida con el resultado de dicha multiplicación. En esta operación, los parámetros de entrada son `10` y `5`, el parámetro de salida es `x`. En el cliente dicha solicitud de servicio se corresponde con la secuencia obligatoria de las acciones `c:s::op(10,5)` y `s:c::op(x)`. Es decir, el cliente tendrá por defecto un evento asociado a la parte *return*. En este caso, la declaración de dicho evento sería `op(Resultado:int)`. En la especificación de la clase del objeto `s` se establece una operación correspondiente al servicio `op(x,y)`. La especificación de la operación en el servidor es:

```
op(X:int,Y: int):
    op = server:client::op(z);
```

Nótese que en la acción `server:client::op(z)` se utilizan los parámetros implícitos **client** y **server** que corresponden al cliente y al servidor de la operación solicitada, `op(x,y)`. Para este ejemplo, la operación `op(x,y)` tendría asociada la evaluación[5]:

[5]Como se dijo anteriormente, una evaluación asociada a una operación es interpretada como una evaluación asociada a la acción de inicio de la operación.

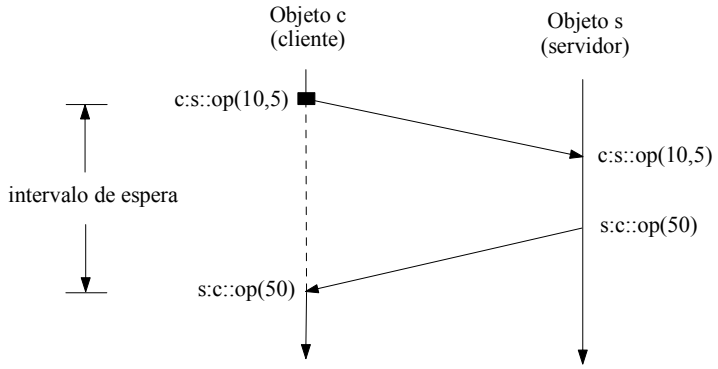

Figura 6.1: Comunicación call/return entre cliente y servidor.

```
[op(X,Y)] z=X*Y;
```

El parámetro de retorno `z` es un atributo en la clase servidora.

La interacción entre los objetos del ejemplo se muestra gráficamente en la Figura 6.1.

Por las características de este tipo de comunicación, el objeto cliente no debe ser a la vez objeto servidor. Es decir, no debería ser una comunicación a **self**.

Ejemplo 55 *Un protocolo en la clase* `socio`*.*

```
socio:
 socio   =  alta.SOCIO1;
 SOCIO1  =  baja + obtener_libro.SOCIO2;
 SOCIO2  =  obtener_libro.SOCIO2 +
            {num_libros>1} devolver_libro.SOCIO2 +
            {num_libros=1} devolver_libro.SOCIO1;
```

Ejemplo 56 *Un protocolo en la clase* `filósofo`*.*

```
filosofo:
 filosofo   =  nacer.FILOSOFO1;
 FILOSOFO1  =  jubilar + pensar.FILOSOFO2;
 FILOSOFO2  =  coger_izq.FILOSOFO3;
 FILOSOFO3  =  coger_der.FILOSOFO4;
 FILOSOFO4  =  comer.FILOSOFO5;
 FILOSOFO5  =  dejar_der.FILOSOFO6;
 FILOSOFO6  =  dejar_izq.FILOSOFO1;
```

Ejemplo 57 *Un protocolo en la clase* ***máquina****. Se trata de una máquina de chocolatinas que acumula hasta tres monedas como crédito.*

```
maquina:
 maquina   =  moneda.MAQUINA1;
 MAQUINA1  =  chocolatina.maquina + moneda.MAQUINA2 +
              ::devolver_moneda.maquina[6];
 MAQUINA2  =  chocolatina.MAQUINA1 + moneda.MAQUINA3 +
              ::devolver_moneda.MAQUINA1;
 MAQUINA3  =  chocolatina.MAQUINA2 +
               ::devolver_moneda.MAQUINA2;
```

Ejemplo 58 *Una operación en la clase* ***máquina*** *del ejemplo anterior.*

```
cancelar:
 cancelar  =  {credito>1}::devolver_moneda.cancelar[7]+
              {credito=1}::devolver_moneda;
```

Ejemplo 59 *Una operación en la clase* `banco`*.*

```
transferencia(Desde:int,Hacia:int,Cantidad:int) transaction:
 transferencia   =  cuenta(numero,[Desde])
                     ::reintegro(Cantidad).TRANSFERENCIA1;
 TRANSFERENCIA1  =  cuenta(numero,[Desde])
                     :reintegro_ok.TRANSFERENCIA2;
 TRANSFERENCIA2  =  cuenta(numero,[Hacia])
                     ::ingreso(Cantidad).TRANSFERENCIA3;
 TRANSFERENCIA3  =  cuenta(numero,[Hacia])
                     :ingreso_ok[8];
```

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

[7] Obligación de realizar un *action sending*.

[8] Obligación de realizar un *action waiting* en el cual el cliente está especificado.

Ejemplo 60 *Operaciones anidadas en la clase* `atm`*, cuyos objetos son cajeros automáticos.*

```
atender:
 atender  =  leer_tarjeta(NumBanco,NumCuenta).ATM1;
 ATM1     =  leer_pin(Pin).ATM2;
 ATM2     =  banco(codigo,[NumBanco])
              ::verificar_pin(NumCuenta,Pin).ATM3;
 ATM3     =  pin_ok.ATM4 + fallo_pin.atender;
 ATM4     =  reintegro(NumBanco,NumCuenta).ATM5 +
             saldo(NumBanco,NumCuenta).ATM5 +
             cancelar.ATM5;
 ATM5     =  ::devolver_tarjeta.atender;

saldo(NumBanco:int,NumCuenta:int):
 saldo  =  banco(codigo,[NumBanco])
            ::obtener_saldo(NumCuenta; Saldo)[9].S1;
 S1     =  ::mostrar_saldo(Saldo);

reintegro(NumBanco:int,NumCuenta:int):
 reintegro  =  leer_monto(Cantidad).R1;
 R1         =  banco(codigo,[NumBanco])
                ::reintegro(NumCuenta,Cantidad).R2;
 R2         =  reintegro_ok.R3 + fallo_reintegro.R4;
 R3         =  ::entregar_dinero(Cantidad);
 R4         =  ::mensaje_fallo;
```

6.3. Clases complejas

Una clase compleja utiliza en su definición otras clases (simples o complejas). Los operadores para construcción de clases complejas son agregación y herencia.

Sintaxis

< def_clase_compleja >	::=	< def_agregación >
		| < def_herencia >

[9] Se trata de una comunicación *call/return*, en la cual el cajero esperará hasta recibir la respuesta desde el banco, para obtener el valor de `Saldo`.

Observaciones

- Independientemente del operador de clases complejas utilizado, la especificación de una clase se hace separadamente de las relaciones entre clases en las cuales pueda participar. Con esto, tanto las clases simples como las clases complejas tienen la misma plantilla de clase. Una clase será compleja si participa (como clase compleja) en alguna relación establecida entre clases mediante algún operador de clases complejas.

- Diremos que una clase compleja tiene propiedades emergentes si tiene propiedades adicionales a las capturadas implícitamente por el operador de clases que la define.

- Si una clase compleja no tiene propiedades emergentes la especificación explícita de su plantilla de clase es opcional.

6.3.1. 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.

Sintaxis

<def_agregación>	::=	<id_clase> **aggregation of** <partes>
<partes>	::=	<sec_def_componente> | <def_asociación>
<def_componente>	::=	[**static**|**dynamic**] [**inclusive**|**relational**] [<id_alias> **alias for**] <id_clase_componente> [**towards** <cardinalidad> [<def_representación>]] [**from** <cardinalidad> [<def_representación>]]

<def_asociación>	::=	<id_class_componente> **grouping by** <lista_id_atributo> [**where '{'**<fórmula>'**}**'] [**towards** <cardinalidad> [<def_representación>]] [**from** <cardinalidad> [<def_representación>]]
<cardinalidad>	::=	'**(**' <card_min> '**,**' <card_max> '**)**'
<def_representación>	::=	**list** '**[**' <def_índice> '**]**' | **set** | **bag**
<def_índice>	::=	<rango> | <lista_cadena>

Observaciones

- **static** implica que los objetos en el componente no pueden ser ni eliminados ni insertados, es decir, los objetos del componente permanecen determinados al crear el objeto agregado. Por el contrario, si de declara **dynamic**, los objetos en el componente pueden cambiar. Si no se especifica **static** o **dynamic**, se asume que es de tipo **dynamic**. Una asociación se considera siempre **dynamic**.

- **inclusive** implica que los objetos del componente sólo pueden interactuar con otros comunicándose a través del objeto agregado. Por el contrario, **relational** significa que los objetos del componente pueden comunicarse directamente con otros, pudiendo no existir coordinación con el objeto agregado. Si no se especifica **inclusive** o **relational**, se asume que es de tipo **relational**. Una asociación se considera siempre **relational**.

- **alias for** permite diferenciar en una agregación entre dos componentes definidos sobre una misma clase.

- **towards** <cardinalidad> representa las cardinalidades vistas desde el objeto agregado hacia los objetos del componente componente. Si no se especifica **towards** <cardinalidad>, se asume que es **towards(0,*)**.

- **from** <cardinalidad> representa las cardinalidades vistas desde un objeto del componente hacia el objeto agregado. Si no se especifica **from** <cardinalidad>, se asume que es **from(0,*)**.

- Debe cumplirse que <card_min> $\leqslant$ <card_max>.

- **list**, **set** y **bag** permiten indicar, de forma opcional y en caso de ser multivaluada (<card_max> > 1), cuál es la representación escogida. En **bag** se permiten elementos duplicados. En **list** la declaración <def_índice> permite definir la forma de referencia usada para acceder a los elementos de la lista. Si no se especifica se asume que es de tipo **set**.

- <def_asociación> permite definir una asociación como una agregación en la cual sólo existe un componente.

- **grouping by** <lista_id_atributo> 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. El mecanismo de identificación por defecto para cada instancia (grupo) se denomina **key** y esta compuesto de la lista de atributos presentes en <lista_id_atributo>.

- En una asociación, al realizarse la agrupación de instancias la especificación **where** permite seleccionar los objetos de la clase del componente que satisfacen la fórmula especificada.

Ejemplo 61 *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’’];
```

Hemos considerado que un `chasis` no puede existir si no está asociado a un `coche`. Las cardinalidades asociadas a este componente delimitan que cada `coche` tiene exactamente un `chasis`. El que sea **static** implica que 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` es diferente del `chasis` en el hecho que 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` (cardinalidad mínima, desde el componente igual a `0`) pero, sólo pueden formar parte de un `coche`. La pertenencia de una `rueda` a un `coche` puede variar con el tiempo (**dynamic**).

Ejemplo 62 *Una especificación alternativa de* `coche` *para el mismo ejemplo anterior podría establecer un componente por cada* `rueda` *del* `coche`*, como se muestra a continuación:*

```
coche aggregation of
    static inclusive chasis
                towards(1,1) from(1,1);
    static inclusive motor
                towards(1,1) from(0,1);
    dynamic inclusive rueda alias adelante_izq
                towards(1,1) from(0,1);
    dynamic inclusive rueda alias adelante_der
                towards(1,1) from(0,1);
    dynamic inclusive rueda alias atras_izq
                towards(1,1) from(0,1);
    dynamic inclusive rueda alias atras_der
                towards(1,1) from(0,1);
    dynamic inclusive rueda alias repuesto
                towards(0,1) from(0,1);
```

Ejemplo 63 *Otra definición para el objeto agregado* `coche`*, modelando separadamente las* `ruedas` *como otra agregación.*

```
coche aggregation of
    static inclusive chasis towards(1,1) from(1,1);
    static inclusive motor towards(1,1) from(0,1);
    static inclusive ruedas towards(1,1) from(0,1);

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

Ejemplo 64 *Una factura y sus líneas.*

```
class factura
identification
    ref:(año,numero);
constant attributes
    año:nat;
    numero:nat;
variable attributes
    pagada:bool(false);
events
    crear new;
    eliminar destroy;
end class

factura aggregation of
    static inclusive encabezado towards(1,1) from(1,1);
    static inclusive lineas towards(1,1) from(1,1);
    static inclusive total towards(1,1) from(1,1);

lineas aggregation of
    dynamic inclusive linea
                towards(1,*) from(1,1) list[1..*];
```

Ejemplo 65 *Asociación de* ***facturas*** *según las definiciones del ejemplo anterior.*

```
facturas_por_año_pendientes aggregation of
    factura grouping by año where {pagada=false};
```

6.3.2. Visibilidad desde el agregado al componente

Independientemente del tipo de agregación, existe visibilidad de atributos del componente desde el agregado. Esto permite utilizar en fórmulas de la clase agregada atributos de clases componentes (prefijando con una referencia al componente). 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 (por existir un morfismo de inclusión fuerte) todas las acciones del componente son también acciones en el agregado (prefijados con una referencia al objeto componente) y pueden ser utilizados en fórmulas de la plantilla de clase agregada. 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.

A continuación se presenta la sintaxis utilizada para hacer referencia a atributos y acciones de componentes en la definición de la clase agregada.

Sintaxis

<atributo_componente>	::=	<ref_objeto>'.' <id_atributo>
		| <ref_objeto>'.'<atributo_componente>
<servicio_componente>	::=	<ref_objeto>'.' <servicio>
		| <ref_objeto>'.' <servicio_componente>

En determinado momento, un objeto agregado tiene asociado como componente un subconjunto del total de instancias de una clase. Así, aunque en la especificación de una agregación <ref_objeto> es una referencia a todos los objetos de una clase, en el contexto de la agregación asumiremos que es una referencia a sólo las instancias de la clase del componente que forman parte del objeto agregado. En el apartado siguiente se muestran ejemplos de este aspecto.

6.3.3. Signatura implícita

Para cada componente y atributo definido existirán funciones y operaciones asociadas disponibles para ser utilizadas en las fórmulas de la plantilla. Estas funciones y operaciones son útiles para manipular componentes (o atributos) cuya cardinalidad es mayor que uno (multivaluados). La siguiente tabla muestra las funciones y operaciones implícitas para atributos y componentes.

FUNCIONES
position(*atributo/componente*,<*referencia*>)
count(*atributo/componente*) [**where** '**{**'<fórmula>'**}**']
min(*atributo/componente*) [**where** '**{**'<fórmula>'**}**']
max(*atributo/componente*) [**where** '**{**'<fórmula>'**}**']
sum(*atributo/componente*) [**where** '**{**'<fórmula>'**}**']
avg(*atributo/componente*) [**where** '**{**'<fórmula>'**}**']
first(*atributo/componente* [, *mecanismo*])
last(*atributo/componente* [, *mecanismo*])
atributo/componente [*índice*]
OPERACIONES
insert(*atributo/componente*,<*referencia*>)
insert(*atributo/componente* [*índice*], <*referencia*>)
remove(*atributo/componente* [*índice*])
remove(*atributo/componente*,<*referencia*>)
remove_all(*atributo/componente*)

Observaciones

- *atributo/componente* es el atributo o componente al cual se le aplica la función u operación.
- Cuando se trata de un objeto componente, <referencia> es una referencia a objeto. En el caso de atributos <referencia> coincide con el valor del *sort* del atributo.
- *mecanismo* es el nombre de un mecanismo de identificación de la clase del componente. Para el caso de un atributo no es necesario.
- La funciones **position**, **first**, **last** y *atributo/componente* [*índice*], así como las operaciones que usan [*índice*], son aplicables sólo a atributos o componentes cuya representación es **list**.

- La función **position** devuelve la posición de un valor o de un objeto en la lista *atributo/componente*.
- La función **count** devuelve el número de valores o instancias en el *atributo/componente*. Cuando *atributo/componente* está representado por **bag**, se cuentan también los duplicados.
- Las funciones **min**, **max**, **sum** y **avg**, permiten calcular el mínimo, el máximo, la suma y el promedio, de los valores de un atributo o valores de atributos de un componente, según corresponda.
- Las funciones **count**, **min**, **max**, **sum** y **avg**, pueden con la especificación **where** restringir los valores o instancias sobre los que se aplican, definiendo una fórmula basada en valores de atributos de la clase agregada o de las clases componentes. Cuando la función se aplica a un atributo en la fórmula asociada a **where** se utiliza la palabra **value** para referirse al valor del atributo.
- **first** (**last**) devuelve el primer (último) valor o mecanismo de identificación en el *atributo/componente*.
- En el caso de una lista las operaciones **insert** y **remove** producen la reorganización de la lista, es decir, no se dejan celdas vacías.
- *índice* es un valor permitido de acuerdo a la definición de índice dada para la lista.

Ejemplo 66 *Consideradas las siguientes partes de una especificación:*

```
...
variable attributes
    notas:nat list[1..*];
...
...
sucursal aggregation of
    dynamic inclusive cuenta alias cuentas_sucursal
                towards(0,*) from(1,1) list[1..*];
...
```

Supondremos que se tiene: `notas[1]=6, notas[2]=8, notas[3]=3` y además, una instancia de sucursal con las siguientes instancias componentes de `cuentas_sucursal`: `cuenta(numero,[100])` y `cuenta(numero,[101])`. Dichas cuentas tienen el valor `10000` y `70000` en su atributo

`balance`, respectivamente. La siguiente tabla muestran ejemplos de funciones y operaciones basadas en las especificaciones anteriores.

Función u operación	**Resultado**
position(notas, 6)	1
count(notas) where {value $>$ 5}	2
first(notas)	6
last(notas)	3
notas[2]	8
position(cuentas_sucursal, cuenta(numero,[101]))	2
sum(cuentas_sucursal.balance) where {balance $>$ 0}	80000
first(cuentas_sucursal)	cuenta(numero,[100])
last(cuentas_sucursal)	cuenta(numero,[101])
cuentas_sucursal[1]	cuenta(numero,[100])
remove(notas, 8)	notas[1]=6, notas[2]=3
insert(notas[1], 9)	notas[1]=9, notas[2]=6, notas[3]=8, notas[4]=3
remove(notas[2])	notas[1]=6, notas[2]=3
remove_all(notas)	$\emptyset$
remove(cuentas_sucursal, cuenta(numero, [100]))	cuentas_sucursal[1]= cuenta(numero, [101])
insert(cuentas_sucursal[1], cuenta(numero, [103]))	cuentas_sucursal[1]= cuenta(numero, [103]), cuentas_sucursal[2]= cuenta(numero, [100]), cuentas_sucursal[3]= cuenta(numero, [101])
remove(cuentas_sucursal[1])	cuentas_sucursal[1]= cuenta(numero, [101])
remove_all(cuentas_sucursal)	$\emptyset$

Ejemplo 67 *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;
    total_gastos:int;
    gastos_1998:int;
derivations
    temperatura_del_coche:=motor.temperatura;
    presion_promedio:=avg(rueda.presion)
                where {rueda.instalada=true};
    total_gastos:=sum(reparacion.importe);
    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 68 *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);
    minimo_numero:= min(numeros);
    promedio_numeros:= avg(numeros);
    primer_numero:= first(numeros);
    ultimo_numero:= last(numeros);
```

6.3.4. Consideraciones adicionales para agregación

Las diferentes posibilidades de agregación tienen consecuencias importantes detalladas a continuación.

1. Consideración asociada a la declaración **inclusive**:

   Sólo se puede acceder a los servicios del componente mediante servicios del objeto agregado.

2. Consideración asociada a la declaración **static:**

   No se dispone de las operaciones implícitas **insert** y **remove** después de la creación del objeto agregado.

3. Consideraciones asociadas a la declaración **towards**(min, max):

- Si $min \geq 1$, junto al evento **new** del agregado deben ocurrir al menos min ocurrencias del evento **insert** asociadas al componente.

- Cada inserción de componente tiene como precondición implícita en el agregado la siguiente:

$$evento\_insert \text{ if } \mathsf{count}(\mathsf{componente}) < \mathsf{max}$$

- Cada vez que se quita un componente se tiene como precondición implícita en el agregado la siguiente:

$$evento\_remove \text{ if } \mathsf{count}(\mathsf{componente}) > \mathsf{min}$$

4. Consideraciones asociadas a la declaración **from**(min,max):

- Si $min \geq 1$, junto al evento **new** del componente deben ocurrir al menos min ocurrencias del *evento_ insert* del determinado componente en objetos agregados.

- Cada inserción del componente en un agregado tiene como precondición implícita que el número de agregaciones en las cuales participa el componente sea menor que max.

- Cada vez que se quita un componente en un agregado se tiene como precondición implícita que el número de agregaciones en las cuales participa el componente sea mayor que min.

6.3.5. 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 eventos específicos 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.

Una especie es una clase como producto cartesiano entre las particiones de más bajo nivel en la jerarquía de clases. Las acciones de creación de instancias son siempre dirigidas a las especies. Consideramos que existe compatibilidad de comportamiento para las particiones estáticas y dinámicas. En cambio, para los roles sólo consideramos compatibilidad de signatura con extensiones tanto horizontales como verticales [31].

Sintaxis

<def_herencia>	::=	<def_partición_estática>
		| <def_partición_dinámica>
		| <def_roles>

Observaciones

- La especificación de una clase se hace separadamente de sus relaciones de herencia existentes. Con esto tanto las clases simples como las clases complejas tienen la misma plantilla de clase. Una clase será compleja si participa (como clase compleja) en alguna relación establecida entre clases utilizando un operador de clases complejas. Así, la relación de especialización (o generalización) por herencia queda especificada independientemente de la plantilla de la clase. Una repercusión importante es en cuanto a las especies. 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.

- No se permiten definiciones circulares por lo que las particiones deben estar formadas por clases que no hayan sido particionadas previamente.

Considerando la compatibilidad de comportamiento en las particiones estáticas y dinámicas, deberá cumplirse lo siguiente:

- Los permisos (precondiciones y protocolos) en las subclases deben implicar lógicamente a los permisos de las superclases. Para que las características fijadas se cumplan se exige que las fórmulas de la superclases "se copien" en la subclase antes de establecer cualquier extensión sobre ellas.

- Las evaluaciones añadidas en las subclases sólo pueden modificar atributos emergentes. Por otro lado, no es posible modificar u olvidar las evaluaciones heredadas desde la superclase.

- Las obligaciones (disparos y operaciones) en la subclase deben implicar lógicamente a las obligaciones de la superclase correspondiente.

- Las condiciones asociadas a las restricciones de integridad en las subclases deben implicar lógicamente a las condiciones asociadas a dichas restricciones en las superclases.

6.3.6. 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.

Sintaxis

<def_partición_estática>	::=	<lista_id_subclase> **static specialization of** <id_clase>

Observaciones

- <lista_id_subclase> son subclases de la misma partición estática sobre la clase determinada por <id_clase>.

Ejemplo 69 *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 especificar propiedades emergentes).

6.3.7. Particiones dinámicas

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.

Sintaxis

<def_partición_dinámica>	::=	<def_dinámica_migración> | <def_dinámica_atributo>

Particiones dinámicas por la ocurrencia de eventos específicos

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.

Sintaxis

<def_dinámica_migración>	::=	<lista_id_subclase> **dynamic specialization of** <id_superclase> **migration relation is** <expresión_migración>
<expresión_migración>	::=	<sec_def_estado>

Observaciones

- <lista_id_subclase> son subclases de la misma partición dinámica de la clase <id_superclase> y <expression_migración> expresa las restricciones asociadas al proceso de migración entre las subclases de la partición dinámica.

- <expresion_migración> se define utilizando la misma sintaxis usada al especificar estados de un proceso (en operaciones y protocolos). Sin embargo, en este caso, los identificadores de estado <def_estado> corresponden a nombres de subclases (las particiones dinámicas involucradas). Las acciones y atributos serán los de la subclase en el correspondiente estado del proceso. El evento de creación de instancias debe estar incluido en la definición de la subclase de entrada en el proceso de migración.

Ejemplo 70 *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 eventos de la signatura de `coche` más, posiblemente, otros de la subclase `funcionando`. Entre ellos está el evento `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`. Desde un punto de vista teórico, el proceso que representa la vida de la instancia de `coche` es la composición de los distintos subprocesos de las subclases y los puntos de enlace entre ellos vienen dados por los eventos indicados en el proceso migratorio.

Ejemplo 71 *Una especialización dinámica de la clase* `persona` *relativo a su estado civil puede ser:*

```
soltero, casado, divorciado, viudo
dynamic specialization of persona migration relation is
            persona = nacer.soltero;
            soltero = casarse.casado;
            casado = divorciarse.divorciado + enviudar.viudo;
            viudo = casarse.casado;
            divorciado = casarse.casado;
```

Tal como ocurría en el ejemplo anterior, los eventos incluidos en la especificación del proceso de migración no pertenecen a la clase `persona`. El evento `nacer` está únicamente en la signatura de `soltero`; el evento `divorciarse` sólo pertenece a `casado`, y así sucesivamente.

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.

Sintaxis

<def_dinámica_atributo>	**::=**	<lista_def_subclase_dinámica> **dynamic specialization of** <id_superclase>
<def_subclase_dinámica>	**::=**	<id_subclase> **where** '**{**'<fórmula>'**}**'

Observaciones

- <id_subclase> son subclases de la misma partición dinámica de la clase <id_superclase> y cada subclase va acompañada de una fórmula definida sobre los atributos de la superclase. La modificación del estado del objeto determina el proceso de migración entre las subclases de la partición dinámica.
- La partición que se haga del conjunto de estados a partir de los valores de los atributos debe ser completa y disjunta.

Ejemplo 72 *Una partición dinámica de la clase* `persona` *en función de los atributos puede ser:*

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

Dicha jerarquía de herencia tiene como atributo relevante la `edad` de la `persona` cuyo valor se calcula como diferencia entre la fecha actual y la fecha de nacimiento de la `persona`. En este caso, independientemente

de las acciones que lleven a la modificación del atributo `edad`, ha sido necesario poder representar la migración en función de dicho atributo y no de eventos que acontezcan.

Ejemplo 73 *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;
```

Ejemplo 74 *Una partición estática de la clase* ***cuenta*** *(fijado en la creación de la misma) puede ser:*

```
cuenta_ahorro,
cuenta_crédito,
cuenta_vivienda static specialization of cuenta;
```

Ejemplo 75 *Algunas especies para el ejemplo anterior, son:*

```
cuenta_ahorro*no_rentable,
cuenta_crédito*muy_rentable,
cuenta_vivienda*no_rentable
```

Un evento 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.

6.3.8. Roles

Mediante roles representamos conductas distintas de un objeto. Una clase puede tener asociada diferentes subclases de rol, cada uno 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.

Sintaxis

<def_roles>	::=	<lista_def_rol> **role of** <id_clase_player>
<def_rol>	::=	<id_clase_rol> [**towards**'('<card_min>','<card_max>')']
		<id_evento_rol>

Observaciones

- <id_clase_rol> son clases del grupo de rol cuya superclase está especificada por <id_clase_player>.
- Con <card_min> y <card_max> indicamos cuántos roles como mínimo y máximo puede desempeñar una instancia de la clase <id_clase_player>. Se cumple que <card_min>≤<card_max>. Si no se especifica **towards** se asume **towards**(0,1).
- El evento <id_evento_rol>, que pertenece a <id_clase_player> determina el comienzo del desempeño del rol (es decir representa el **new** en dicha clase de rol).
- La destrucción del objeto que representa el rol no implica la destrucción del objeto player.
- Añadir una clase de rol no genera nueva especies. Sin embargo, particiones de dicha clase de rol sí que implican a nuevas especies.

Ejemplo 76 *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 77 *Consideremos la partición dinámica de* ***persona*** *junto a los roles definidos sobre la misma clase en el ejemplo anterior, es decir:*

```
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 entra a desempeñar el rol. En cambio, a futuras particiones (estáticas o dinámicas) de la clase de rol sí formarían especies entre sí.

Ejemplo 78 *Consideremos que junto a la clase de rol* ***estudiante*** *tenemos definidas las siguientes particiones:*

```
mal_estudiante where {nota_media<5},
buen_estudiante where {nota_media>=5}
    dynamic specialization of estudiante;

extranjero,
no_extranjero
    static specialization of estudiante;
```

Ejemplo 79 *Algunas posibles especies para el ejemplo anterior son:*

```
mal_estudiante*extranjero,
buen_estudiante*no_extranjero, etc.
```

Ejemplo 80 *Si se desea que una* ***persona*** *desempeñe el rol de* ***empleado*** *y de* ***estudiante*** *de forma simultánea separamos la definición de cada grupo de rol:*

```
estudiante towards(0,1) ser_matriculado role of persona


empleado towards(0,10) ser_contratado role of persona
```

Una subclase de rol debe definir los mecanismos de identificación de los roles de manera que sea posible distinguirlos aun cuando sean desempeñados por el mismo objeto.

6.3.9. Especies y herencia múltiple en OASIS

El producto entre las clases hoja de la jerarquía de herencia proporciona el conjunto de especies que reúnen las propiedades de cada una de las clases involucradas. Cualquier otra propiedad emergente debe ser indicada en la especificación de la plantilla de clase. Si todas las clases heredan de una superclase concreta entonces tenemos por definición cualquier combinación de especies con herencia múltiple.

Ejemplo 81 *La clase* `camión*diesel` *puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Dicho atributo debe añadirse a la especificación de dicha clase. Para ello, especificaremos la plantilla de la clase* `camión*diesel`*, como para el resto de clases.*

Evidentemente, para aquellas clases derivadas por herencia múltiple y sin propiedades emergentes no será preciso definir su plantilla .

6.3.10. 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 adicionales a los por defecto heredados.

La creación de un nuevo objeto debe hacerse dentro de la especie a la que va a pertenecer. Como hemos comentado, una especie involucra a una o más clases que pueden provenir bien de particiones estáticas, bien de dinámicas. El evento de creación va dirigido pues a esta especie intersección de todas en las que el objeto que va a ser creado se especializa. La cuestión es si se pueden dar por supuestas determinadas especializaciones y no tener que ser indicadas de forma explícita. Consideraremos los siguientes casos:

Caso 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.

Ejemplo 82 *Para el caso anterior de la clase* `coche`*:*

```
camión,
coche,
otro_vehiculo
    static specialization of vehículo;

gasolina,
diesel,
otro_tipo
    static specialization of vehículo;
```

Los eventos de creación deben dirigirse a las especies completas, es decir a `camión*diesel, coche*gasolina`, etc.

Caso 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:

1. 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.

2. Aquellas subclases de especializaciones dinámicas cuyo proceso migratorio está definido sobre los eventos de migración. En este caso si sólo existe un punto de entrada en el proceso migratorio, el evento de creación será el indicado 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, el evento de creación será el incluido en la subclase dinámica indicada.

En la especificación de la clase *player,* para el caso de los roles, debe existir el evento que determina el desempeño del rol. Este evento es el que se especificó al definirse el grupo de rol correspondiente. La ejecución de dicho evento corresponde a la creación de la instancia de rol. La especificación de la clase de rol puede poseer un evento de destrucción del rol que determina la finalización del rol[10]. En el caso en que el *player*

[10]Que nada tiene que ver con el evento de destrucción de la clase *player*.

ejecute de nuevo el evento de creación del rol, previa a la finalización del actualmente en curso, el objeto tendrá dos roles de la misma clase de rol simultáneamente (siempre que las restricciones de cardinalidad lo permitan). Por definición, si el *player* deja de existir, todos sus roles también serán destruidos.

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

6.4. 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.
- 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.

Existen acciones que no son forzadas a ocurrir mediante obligaciones definidas en las plantillas de clase. Estas acciones corresponden a las conexiones del sistema con el mundo real (el entorno) y representan la frontera con los objetos reales, siendo el límite a partir del cual el diseñador se abstrae de la dependencia causal que origina una cierta obligación. Por ejemplo, cuando el socio de una biblioteca devuelve un libro; cuando el cliente inserta la tarjeta en un cajero automático; cuando un

sensor envía cierta señal al sistema, etc. En estos casos, aunque no exista una definición explícita en la plantilla de clase del cliente, debe estar definida dicha relación de acceso en alguna interfaz.

Sintaxis

<def_interfaz>	::=	**interface** <ref_cliente> **with** <ref_servidor> [**attributes** '**(**' <atributos_ofrecidos> '**)**'] **services** '(' <servicios_ofrecidos> '**)**' **end interface**
<ref_cliente>	::=	<ref_objeto>
<ref_servidor>	::=	<ref_objeto>
<atributos_ofrecidos>	::=	**all** | **none** | <lista_id_atributo>
<servicios_ofrecidos>	::=	**all** | **none** | <lista_id_servicio>

Observaciones

- Cuando no se define **attributes** se asume que no se ofrecen atributos, excepto en el caso cuando <ref_servidor> es **self**, en el cual se asume que se ofrecen todos los atributos.
- Si para un cliente puede aplicarse más de una interfaz respecto de un determinado objeto servidor, prevalecen las definiciones de la interfaz más especifica (más cercana a los objetos cliente y servidor involucrados).
- 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.

Ejemplo 83 *Una interfaz entre un* `cliente` *y un* `cajero automático`*.*

```
interface cliente(someone) with cajero_automatico(someone)
    attributes(fuera_de_servicio)
```

```
    services(insertar_tarjeta,
        consultar_saldo,retirar_dinero,cancelar)
end interface
```

Ejemplo 84 *Una interfaz entre una* `cuenta` *bancaria y sí misma.*

```
interface cuenta(someone) with self
    services(aplicar_intereses,enviar_informe)
end interface
```

Ejemplo 85 *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
```

6.4.1. Niveles de interfaz

La interfaz puede establecer relaciones de acceso entre objetos y/o conjuntos de objetos, dependiendo de si <ref_objeto> es una referencia a un objeto individual, grupo de objetos de una clase o todos los objetos de una clase. Así, las interfaces pueden ser definidas en cuatro niveles diferentes de detalle, que se describen a continuación.

Interfaz clase-clase. Es una interfaz en la cual ambos, <ref_cliente> y <ref_servidor> se definen usando la sintaxis <id_clase>(someone).

Ejemplo 86 *Una interfaz entre cualquier objeto de la clase* `usuario` *y cualquier objeto de la clase* `cuenta`*.*

```
interface usuario(someone) with cuenta(someone)
    ...
end interface
```

Interfaz objeto-clase. Es una interfaz en la cual <ref_cliente> se refiere a un objeto en particular y <ref_servidor> se define usando la sintaxis <id_clase>(someone).

Ejemplo 87 *Una interfaz entre el objeto de la clase* `usuario` *identificado por* `nombre=''Juan Pérez''` *y cualquier objeto de la clase* `cuenta`*.*

```
interface usuario(nombre,[''Juan Perez''])
                with cuenta(someone)
    ...
end interface
```

Interfaz clase-objeto. Es una interfaz en la cual <ref_cliente> se define usando la sintaxis <id_clase>(someone) y <ref_servidor> se refiere a un objeto en particular.

Ejemplo 88 *Una interfaz entre un objeto de la clase* `usuario` *y el objeto de la clase* `cuenta` *identificado por* `número=100`*.*

```
interface usuario(someone) with cuenta(numero,[100])
    ...
end interface
```

Interfaz objeto-objeto. Es una interfaz en la cual ambos, <ref_cliente> y <ref_servidor> se refieren a un objeto en particular.

Ejemplo 89 *Una posible interfaz entre el objeto de la clase* `usuario` *que está identificado por* `nombre=''Juan Pérez''` *usuario y el objeto de la clase* `cuenta` *identificado por* `número=100`*.*

```
interface usuario(nombre,[''Juan Pérez''])
                with cuenta(numero,[100])
    ...
end interface
```

Capítulo 7

Ejemplo OASIS 3.0

7.1. Descripción del ejemplo

Se desea obtener el modelo conceptual de un sistema de gestión de tarjetas de crédito. Las tarjetas tienen un titular y son dadas de alta por vendedores de las sucursales emisoras de dichas tarjetas. Las tarjetas tienen una fecha de caducidad después de la cual pasarán a estar bloqueadas. El administrador de la sucursal de tarjetas puede bloquear una tarjeta cuando lo considere oportuno. Cada tarjeta está asociada a una cuenta de crédito. En dicha cuenta de crédito se cargarán los gastos que los titulares de tarjetas realicen en las distintas tiendas. Mediante una tarjeta que no esté bloqueada una tienda puede registrar el pago de una compra. El cliente decidirá el número de cuotas y los meses en los que desea fraccionar el pago del importe de dicha compra. La suma de las cuotas debe ser el importe de la compra. Una cuota particular de una compra puede pagarse de varias formas. La primera, y más normal, es a través de una transferencia desde una cuenta corriente de alguna sucursal bancaria. Cada pago de cuota puede hacerse desde una cuenta corriente distinta. Una segunda forma de pago es manualmente, sin intervenir la transferencia entre cuentas y se realiza cuando el empleado de las tarjetas recibe en metálico la cuota relativa a dicho mes. Una cuota puede ser cancelada sin contabilizarse a efectos de pagada ni sobre el total del importe de compra. Dado que una cuota no puede ser eliminada del sistema, la cancelación es la forma de anular cuotas introducidas de forma errónea. El importe de la cuenta en cualquier instante debe ser igual a la suma de las cantidades asociadas a las cuotas respectivas. Con esto,

una compra por un importe de 800 EUROS puede introducirse en tres cuotas de 200, 100 y 500, siendo el total de compra a cada paso de 200, 300 y 800. Las tiendas se agrupan por distritos necesitándose información estadística relativa a los mismos. Un distrito cuyo acumulado de compras sea superior a 10000 EUROS se considerará distrito comercial. Aquellos distritos no comerciales podrán solicitar de su respectivo ayuntamiento una subvención pública para promocionar las compras en sus tiendas. También se desea mantener información de las cuotas existentes en un mes concreto del año y de las cuotas de un mes pero asociadas a una cuenta de crédito particular. Se desea mantener listas de clientes de tarjetas que se consideran favoritos.

7.2. Especificación en OASIS 3.0

A continuación se presenta la especificación en OASIS 3.0 para el ejemplo planteado. En esta especificación aparecen aspectos no incluidos en la descripción del problema y que han sido incorporados para mostrar algunos detalles de la expresividad del lenguaje.

```
conceptual schema tarjetas_de_credito
class usuario[1]
identification
    userid:(userid);
constant attributes
    userid:string;
    nombre:string towards(1,1);
variable attributes
    password:string;
events
    alta new;
    baja destroy;
    cambiar_password(texto:string);
valuations
    [cambiar_password(texto)] password:=texto;
end class
```

[1]Superclase de las clases `administrador_sistema`, `administrador_tarjeta`, `vendedor_tarjeta`, `empleado_tarjeta` y `tienda`.

```
class titular
identification
    dni:(dni);
constant attributes
    dni:string;
    nombre:string towards(1,1);
    firma:blob towards(1,1);
    foto:blob towards(1,1);
variable attributes
    email:string;
    numeros_telefono:string towards(0,*) list[1..*];
events
    alta new;
    baja destroy;
    fijar_email(texto:string);
    añadir_telefono(texto:string) alias for
                        insert(numeros_telefono[1],texto);
valuations
    [fijar_email(texto)] email:=texto;
end class
```

```
class cuenta_credito
identification
    numero:(numero);
constant attributes
    numero:nat;
    fecha_apertura:date towards(1,1);
variable attributes
    limite_mensual:int towards(1,1);
    suma_acumulada_pagos:int(0) towards(1,1);
events
    alta new;
    baja destroy;
    pagar_cuota(cant:int);
    fijar_limite(cant:int);
valuations
    [pagar_cuota(cant)] suma_pagos:=suma_acumulada_pagos+cant;
    [fijar_limite(cant)] limite_mensual:=cant;
end class
```

```
class cuenta_cte
identification
    numero:(numero);
constant attributes
    numero:nat;
variable attributes
    pagos:int(0) towards(1,1);
events
    alta new;
    baja destroy;
    extraer(cant:int);
valuations
    [extraer(cant)] pagos:=pagos+cant;
end class
```

```
class tarjeta
identification
    codigo:(codigo);
constant attributes
    codigo:string;
    fecha_caducidad:date towards(1,1);
events
    alta new;
end class
```

[2]

```
class banco
identification
    nombre:(nombre);
constant attributes
    nombre:string;
variable attributes
    direccion:string;
events
    alta new;
    baja destroy;
    fijar_direccion(direc:string);
valuations
    [fijar_direccion(direc)] direccion:=direc;
end class
```

[3]

[2] Superclase de `tarjeta_activa` y `tarjeta_bloqueada`.

[3] Agregación con componente estricto `sucursal_banco`.

```
_____________________
class cuota_mensual
identification
    ref:(año,mes);
constant attributes
    año:nat;
    mes:nat;
    importe_cuota:int towards(1,1);
variable attributes
    pagada:bool(false) towards(1,1);
    cancelada:bool(false) towards(1,1);
events
    alta new;
    pagarse(cant:int);
    pagarse_manualmente(cant:int);
    cancelarse;
valuations
    [pagarse(cant)] pagada:=true;
    [pagarse_manualmente(cant)] pagada:=true;
    [cancelarse] cancelada:=true;
preconditions
    pagarse(cant) if {importe_cuota=cant};
    pagarse_manualmente(cant) if {importe_cuota=cant};
protocols
    pago_cancelacion:
     pago_cancelacion  =  pagarse(cant) +
                          pagarse_manualmente(cant) +
                          cancelarse;
end class

_______________________
class vendedor_tarjeta
identification
    codigo:(codigo);
constant attributes
    codigo:string;
    nombre:string towards(1,1);
events
    alta new;
    baja destroy;
end class
```

```
class entidad_publica
identification
    ref_entidad:(distrito);
constants attributes
    distrito:nat;
events
    alta new;
    baja destroy;
    solicitar_subvencion;
end class
```

```
class tienda
identification
    cif:(cif);
constant attributes
    cif:string;
    nom_fiscal:string towards(1,1);
    distrito:string towards(1,1);
events
    alta new;
    baja destroy;
end class
```

```
class compra
identification
    codigo:(codigo);
constant attributes
    codigo:nat;
    fecha:date towards(1,1);
    hora:time towards(1,1);
    numero_cuotas:nat(1) towards(1,1);
variable attributes
    importe:nat(0) towards(1,1);
events
    alta new;
    añadir_importe(cant:real);
    restar_importe(cant:real);
valuations
    [añadir_importe(cant)]importe:=importe+cant;
    [restar_importe(cant)]importe:=importe-cant;
end class
```

```
class sucursal[4]
identification
    nombre:(nombre);
constant attributes
    nombre:string;
variable attributes
    direccion:string;
    numeros_telefono:string towards(0,*) list[1..*];
events
    alta new;
    baja destroy;
    fijar_direccion(direc:string);
    añadir_telefono(texto) alias for
                        insert(numeros_telefono[1],texto);
valuations
    [fijar_direccion(direc)] direccion:=direc;
end class
```

```
class sucursal_tarjetas
identification
    nombre_tarjeta:(nombre_tarjeta);
constant attributes
    nombre_tarjeta:string;
end class
```

```
class distrito_no_comercial[5]
variable attributes
    solicitado:bool(false) towards(1,1);
valuations
    [entidad_publica(ref_entidad,[distrito])
        ::solicitar_subvencion] solicitado=true
triggers
    entidad_publica(ref_entidad,[distrito])
        ::solicitar_subvencion
            when {solicitado=false};
end class
```

[4]Superclase de `sucursal_tarjetas` y `sucursal_banco`.
[5]Subclase de `compras_X_tienda_por_distrito`.

```
class tarjeta_activa[6]
variable attributes
    ultima_fecha_uso:date;
events
    baja destroy;
    bloquear_tarjeta[7];
    usar_tarjeta(fecha:date);
valuations
    [usar_tarjeta(fecha)] ultima_fecha_uso:=fecha;
triggers
    self::bloquear_tarjeta if {currenttime[8]>fecha_caducidad};
end class

class tarjeta_X_compra[9]
identification
    ref:(tarjeta.codigo,compra.codigo);
events
    nueva_compra new sharing with members
            tarjeta_activa(tarjeta.codigo,[tarjeta.codigo]).
                usar_tarjeta(compra.fecha),
            compra.alta;
end class

class cuentas_credito_X_vendedor_tarjeta[10]
identification
    ref:(vendedor_tarjeta.codigo);
derived attributes
    cuantas_ha_creado:nat;
derivations
    cuantas_ha_creado:=count(cuenta_credito);
end class
```

[6] Subclase de `tarjeta`.

[7] Evento que produce la migración de una instancia de la subclase `tarjeta_activa` hacia la subclase `tarjeta_bloqueada`.

[8] Atributo implícito de dominio `time` que representa la hora y fecha actual del objeto.

[9] Agregación cuyos componentes `tarjeta` y `compra` no permiten nulos. Además, el componente `compra` es estricto.

[10] Agregación con componentes estrictos `cuenta_credito` y `vendedor`. Su componente `vendedor` no permite nulos.

```
class cuentas_cte_X_sucursal_banco[11]
identification
    ref:(sucursal_banco.nombre);
derived attributes
    num_cuentas_sucursal_banco:int;
derivations
    num_cuentas_sucursal_banco:=count(cuenta_cte);
end class
```

```
class cuotas_mensual_X_cuenta_credito[12]
identification
    ref:(cuenta_credito.numero).
derived attributes
    cuantas_no_pagadas:int;
    cuantas_pagadas:int;
derivations
    cuantas_no_pagadas:=count(cuota_mensual)
        where {cuota_mensual.pagada=false
            or cuota_mensual.cancelada=true};
    cuantas_pagadas:=count(cuota_mensual)-cuantas_no_pagadas;
events
    pagarse_manualmente(valoraño:nat,valormes:nat,
            valorimporte_cuota:real)
        sharing with members
            cuota_mensual(ref,[valoraño,valormes]).
                pagarse_manualmente(valorimporte_cuota);
            cuenta_credito.pagar_cuota(valorimporte_cuota);
    pagar_cuota_cuenta(valoraño,valormes,valorimporte_cuota)
        sharing with members
            cuota_mensual(ref,[valoraño,valormes]).
                pagarse(valorimporte_cuota),
            cuenta_credito.pagar_cuota(valorimporte_cuota);
    cancelar_cuotas_cuenta
        calling to members cuota_mensual.cancelarse;
end class
```

[11] Agregación cuyos componentes `cuenta_cte` y `sucursal_banco` son estrictos. Además `sucursal_banco` no permite nulos.

[12] Agregación cuyos componentes `cuota_mensual` y `cuenta_credito` son estrictos y no permiten nulos.

```
class cuotas_mensual_X_cuenta_credito_X_compra[13]
identification
    ref:(compra.codigo);
derived attributes
    suma_cuotas_pendientes:int;
    numero_cuotas_pendientes:int;
    total_de_cuotas:int;
    total_importes:int;
    importe:int;
    numero_cuotas:int;
derivations
    suma_cuotas_pendientes:=sum(
            cuotas_mensual_X_cuenta_credito.
            cuota_mensual.importe_cuota)
        where {pagada=false and cancelada=false};
    numero_cuotas_pendientes:=count(
            cuotas_mensual_X_cuentacredito.cuota_mensual)
             where {pagada=false and cancelada=false};
    total_de_cuotas:=count(cuotas_mensual_X_cuentacredito.
            cuota_mensual);
    total_importes:=sum(cuotas_mensual_X_cuenta_credito.
            cuota_mensual.importe_cuota)
        where {cancelada=false};
    importe:=compra.importe;
    numero_cuotas:=compra.numero_cuotas;
constraints
    numero_cuotas=total_de_cuotas;
    importe=total_importes;
events
    cancelar_cuota_compra(valoraño:nat,valormes:nat)
        sharing with members
            compra.restar_importe(cuotas_cuentacredito.
                cuota_mensual(ref,[valoraño,valormes]).
                    importe_cuota),
            cuotas_mensual_X_cuenta_credito.
                cuota_mensual(ref,[valoraño,valormes]).
                    cancelarse;
```

[13] Agregación cuyos componentes `cuotas_mensual_X_cuenta_credito` y `compra` son estrictos y no permiten nulos.

```
    añadir_cuota_compra
        sharing with members
            compra.añadir_importe(
                    cuotas_mensual_X_cuenta_credito.
                    cuota_mensual.importe_cuota),
            cuotas_mensual_X_cuenta_credito.
                    cuota_mensual.alta;
end class
```

```
class clientes_favoritos[14]
identification
    nombre_lista(nombre);
constant attributes
    nombre:string;
derived attributes
    cuantos:nat;
    edad_mas_joven:nat;
    dni_del_primero:string;
    nombre_del_primero:string;
derivations
    cuantos:=count(titular);
    edad_mas_joven:=min(titular.edad);
    dni_del_primero:=first(titular.dni);
    nombre_del_primero:=last(titular.nombre);
events
    crear_lista new;
    borrar_lista destroy;
    añadir(valordni:nat) alias for
                insert(dni[1],[valordni]);
    suprimir alias for remove(dni[1]);
preconditions
    borrar_lista if {cuantos=0}
end class
```

[14] Agregación sobre `titular`.

```
_____________________
class transferencia[15]
identification
    ref:(cuenta_cte.numero,cuenta_credito.numero,
            año,mes,fecha);
constant attributes
    año:nat;
    mes:nat;
    fecha:date;
events
    transferir
        new sharing with members
            cuenta_cte(numero,[cuenta_cte.numero]).
                    extraer(cant),
            cuotas_mensual_X_cuenta_credito(ref,
                        [cuenta_credito.numero]).
                            pagar_cuota_cuenta(año,mes,cant);
end class

_____________________________________
class cuotas_mensual_por_año_mes[16]
identification
    ref:(año,mes);
derived attributes
    suma_cuotas:int;
    media_cuotas_millonarias:int;
    hay_cuotas_millonarias:bool;
    cuota_mayor:int;
    cuota_menor:int;
derivations
    suma_cuotas:=sum(importe_cuota);
    media_cuotas_millonarias:=avg(importe_cuota)
        where {cantidad>=1000};
    hay_cuotas_millonarias:=count(cuota_mensual)
        where {importe_cuota>=1000};
    cuota_mayor:=max(importe_cuota);
    cuota_menor:=min(importe_cuota);
end class
```

[15] Agregación en la cual no se permiten nulos para los componentes: `cuenta_cte` y `cuotas_mensual_X_cuenta_credito`.

[16] Agregación cuyo componente `cuota_mensual` es estricto.

```
class compras_X_tienda_por_distrito[17]
identification
    distrito:(tienda.distrito);
constants attributes
    nombre_responsable:string towards(1,1);
    direccion_responsable:string;
derived attributes
    distrito:string;
    suma_compras:int;
    media_compras_millonarias:int;
    hay_compras_millonarias:nat;
    compra_mayor:int;
    compra_menor:int;
derivations
    distrito:=tienda.distrito;
    suma_compras:=sum(compra.importe);
    media_compras_millonarias:=avg(compra.importe)
        where {compra.importe>=1000};
    hay_compras_millonarias:=count(compra)
        where {compra.importe>=1000};
    compra_mayor:=max(compra.importe);
    compra_menor:=min(compra.importe);
end class
```

```
class cuotas_mensual_X_cuenta_credito_por_año_mes_numero[18]
identification
    ref:(codigo,año,mes);
derived attributes
    suma_cuotas:int;
    limite:int;
derivations
    suma_cuotas:=sum(cuota_mensual.importe_cuota);
    limite:=cuenta_credito.limite_mensual;
constraints
    {suma_cuotas<=limite};
end class
```

[17] Agregación cuyo componente `compras_X_tienda` es estricto y no permite nulos.
[18] Agregación cuyo componente `cuotas_mensual_X_cuenta_credito` es estricto.

```
administrador_sistema,
administrador_tarjeta,
vendedor_tarjeta,
empleado_tarjeta, tienda
    static specialization of usuario;


banco aggregation of
    dynamic relational sucursal_banco towards(0,*) from(1,1);


tarjeta_activa,
tarjeta_bloqueada
    dynamic specialization of tarjeta
    migration relation is
        tarjeta=alta.tarjeta_activa
        tarjeta_activa=bloquear_tarjeta.tarjeta_bloqueada
        tarjeta_bloqueada=desbloquear_tarjeta.tarjeta_activa;


sucursal_tarjetas,
sucursal_banco
    static specialization of sucursal;

distrito_comercial where {suma_compras>10000},
distrito_no_comercial where {suma_compras<=10000}
    dynamic specialization of compras_X_tienda_por_distrito;

tarjeta_X_compra aggregation of
    static relational tarjeta towards(1,1) from(0,*);
    static relational compra towards(1,1) from(1,1);


cuentas_credito_X_vendedor_tarjeta aggregation of
    dynamic relational cuenta_credito towards(1,*) from(1,1);
    static relational vendedor_tarjeta towards(1,1) from(0,*);

cuentas_cte_X_sucursal_banco aggregation of
    dynamic relational cuenta_cte towards(0,*) from(1,1);
    static relational sucursal_banco towards(1,1) from(0,1);

cuotas_mensual_X_cuenta_credito aggregation of
    dynamic relational cuota_mensual
        towards(1,*) from(1,1) list[1..*];
    static relational cuenta_credito towards(1,1) from(0,1);
```

```
cuotas_mensual_X_cuenta_credito_X_compra aggregation of
    static relational cuotas_mensual_X_cuenta_credito
        towards(1,1) from(1,1);
    static relational compra towards(1,1) from(1,1);

transferencia aggregation of
    static relational cuenta_cte towards(1,1) from(0,*);
    static relational cuotas_mensual_X_cuenta_credito
        towards(1,1) from(0,1);


compras_X_tienda_por_distrito aggregation of
    compra_X_tienda grouping by tienda.distrito
        towards(0,*) from(1,1);

clientes_favoritos aggregation of
    titular towards(0,*) from(0,1) list[1..*];

cuotas_mensual_por_año_mes aggregation of
    cuota_mensual grouping by año,mes towards(1,*) from(1,1);

cuotas_mensual_X_cuenta_credito_por_año_mes_numero
    aggregation of
        cuotas_mensual_X_cuenta_credito grouping by año,mes,
        cuenta_credito.numero towards(1,*) from(1,1);

tarjetas_X_cuenta_credito aggregation of
    dynamic relational tarjeta towards(1,*) from(1,1);
    static relational cuenta_credito towards(1,1) from(1,1);

compra_X_tienda aggregation of
    static relational compra towards(1,1) from(1,1);
    static relational tienda towards(1,1) from(0,*);


cuentas_credito_X_sucursal_tarjetas aggregation of
    dynamic relational cuenta_credito towards(1,*) from(1,1);
    static relational sucursal_tarjetas
            towards(1,1) from(0,1);


cuenta_credito_X_titular aggregation of
    dynamic relational cuenta_credito towards(1,1) from(1,1);
    static relational titular towards(1,1) from(1,*);
```

```
interface administrador_sistema(someone)
    with usuario(someone)
    attributes(all)
    services(alta)
end interface

interface administrador_sistema(someone)
    with self[19]
    attributes(all)
    services(none)
end interface

interface vendedor_tarjeta(someone)
    with titular(someone)
    attributes(all)
    services(alta)
end interface

interface administrador_tarjeta(someone)
    with titular(someone)
    attributes(all)
    services(baja)
end interface

interface empleado_tarjeta(someone)
    with titular(someone)
    attributes(all)
    services(fijar_email,añadir_telefono)
end interface

interface vendedor_tarjeta(someone)
    with cuenta_credito(someone)
    attributes(all)
    services(alta)
end interface

interface administrador_tarjeta(someone)
    with cuenta_credito(someone)
    attributes(all)
    services(baja)
end interface
```

[19] Esta interfaz prevalece sobre la anterior cuando el servidor es el mismo objeto cliente (`administrador_sistema`). Su propósito es impedir que un objeto de la clase `administrador_sistema` pueda darse de baja a sí mismo.

```
interface empleado_tarjeta(someone) with cuenta_cte(someone)
    attributes(all)
    services(alta)
end interface

interface administrador_tarjeta(someone)
    with cuenta_cte(someone)
    attributes(all)
    services(baja)
end interface

interface administrador_tarjeta(someone)
    with vendedor_tarjeta(someone)
    attributes(all)
    services(alta,baja)
end interface

interface administrador_tarjeta(someone)
    with empleado_tarjeta(someone)
    attributes(all)
    services(alta,baja)
end interface

interface empleado_tarjeta(someone)
    with tarjeta(someone)
    attributes(all)
    services(alta)
end interface

interface empleado_tarjeta(someone)
    with banco(someone)
    attributes(all)
    services(alta,baja,fijar_direccion)
end interface

interface empleado_tarjeta(someone)
    with entidad_publica(someone)
    attributes(all)
    services(alta,baja)
end interface

interface distrito_no_comercial(someone)
    with entidad_publica(someone)
    services(solicitar_subvencion)
end interface
```

```
interface administrador_tarjeta(someone)
    with tienda(someone)
    attributes(all)
    services(alta,baja)
end interface

interface empleado_tarjeta(someone)
    with sucursal(someone)
    attributes(all)
    services(alta,baja,fijar_direccion,añadir_telefono)
end interface

interface administrador_tarjeta(someone)
    with tarjeta_activa(someone)
    attributes(all)
    services(baja,bloquear_tarjeta)
end interface

interface tienda(someone)
    with tarjeta_X_compra(someone)
    attributes(all)
    services(nueva_compra)
end interface

interface empleado_tarjeta(someone)
    with cuotas_mensual_X_cuenta_credito(someone)
    attributes(all)
    services(pagarse_manualmente,cancelar_cuotas_cuenta)
end interface

interface tienda(someone)
    with cuotas_mensual_X_cuenta_credito_X_compra(someone)
    attributes(all)
    services(cancelar_cuota_compra,añadir_cuota_compra)
end interface

interface empleado_tarjeta(someone)
    with transferencia(someone)
    attributes(all)
    services(transferir)
end interface

interface administrador_tarjeta(someone)
    with clientes_favoritos(someone)
    attributes(all)
```

```
    services(crear_lista,borrar_lista,añadir,suprimir)
end interface
end conceptual schema
```

Capítulo 8

Conclusiones

Este documento presenta la versión 3.0 de OASIS. Respecto de la versión anterior se destacan las siguientes mejoras:

- Utilización de Lógica Dinámica (una variante de la misma, en la que se codifican además aspectos de Lógica Deóntica) como marco formal uniforme que cubre todas las secciones de la especificación de una clase. Desde esta perspectiva global, las fórmulas asociadas a una especificación OASIS representan: alcance entre mundos, permiso de ocurrencia de acciones u obligación de ocurrencia de acciones.

- 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 (*operations*) y procesos de prohibición (*protocols*). Las operaciones permiten modelar un comportamiento algorítmico, asociado a un servicio no atómico provisto por el objeto. Los protocolos constituyen una extensión al concepto de *process* en la anterior versión.

- Enriquecimiento de los mecanismos usados para definir clases complejas: agregación y herencia. En agregación se ha caracterizado con mayor detalle las propiedades asociadas a los distintos tipos

de agregación. En herencia se ha introducido una visión más amplia, incluyendo tres formas de herencia: 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.

OASIS provee un enfoque formal para la especificación del modelo conceptual de sistemas de información. El objetivo de OASIS es poder expresar los requisitos funcionales de un sistema de información en un marco formal que facilite su validación, verificación y generación automática de programas.

El modelo subyacente de OASIS ha sido caracterizado definiendo un lenguaje de especificación. La BNF del lenguaje (su sintaxis) unida a las reglas semánticas presentadas, proporciona una visión detallada del modelo OO subyacente.

Una representación textual del modelo conceptual en OASIS no es la forma más adecuada para llevar a cabo el análisis y diseño del sistema. Tampoco es directo el proceso de obtener una implementación (en algún grado) a partir de una especificación OASIS textual. Sin embargo, el metamodelo subyacente a una especificación OASIS corresponde al repositorio de una herramienta CASE, pieza clave para un proceso de desarrollo automatizado de sistemas de información. Esta automatización incluye esencialmente las siguientes características: integración de herramientas gráficas y ventanas de diálogo que faciliten la edición del modelo del sistema, verificación de reglas semánticas del metamodelo, capacidades de animación automática y verificación del modelo, síntesis de programas (en algún grado) en la plataforma objetivo, etc.

¿Qué beneficios aporta el utilizar OASIS como modelo subyacente en un entorno CASE?. Una especificación OASIS se interpreta formalmente como una presentación en Lógica Dinámica. Esta correspondencia provee propiedades al sistema modelado. Dichas propiedades pueden ser utilizadas para: animación automática de la especificación (mediante la construcción de un motor que ejecute las fórmulas correspondientes), análisis de la especificación y verificaciones de consistencia complejas, y finalmente, establecimiento de pautas robustas para la síntesis de programas (determinando correspondencias entre fórmulas en Lógica Dinámica y elementos del lenguaje objetivo).

¿Puede la edición textual de una especificación OASIS ser totalmente reemplazada por la edición a través de herramientas gráficas?. Esto parece difícil de conseguir sin perder la claridad de una representación gráfica (su principal ventaja). Por eso, el entorno CASE tiene que proveer, de forma integrada las herramientas gráficas y el uso de ventanas de diálogo para editar toda la información del modelo que no sea conveniente editar gráficamente.

De acuerdo con lo planteado, los trabajos de investigación actuales están enfocados principalmente en los siguientes aspectos:

- Síntesis de programas en entornos de desarrollo de software industrial.
- Animación automática de especificaciones OASIS en entornos concurrentes, en particular usando Redes de Petri [27] y Programación Lógica Concurrente [19] de acuerdo con el modelo de ejecución establecido en [17]. Este trabajo se orienta a la especificación incremental y validación temprana de requisitos.
- Extensiones al modelo OASIS, aumentando su capacidad expresiva.
- Formalización del proceso de desarrollo y evolución del software desde la perspectiva del modelo OASIS [3].
- Desarrollo y adaptación de herramientas que, manteniendo implícito el modelo OASIS, sean capaces de capturar requisitos con una notación más cercana al espacio del problema.
- Todos los aspectos anteriores se enmarcan dentro del contexto del refinamiento y extensión de OO-METHOD y del entorno CASE para la producción de software, que ya ha sido desarrollado.

Bibliografía

[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] Canos J.H. OASIS: Un lenguaje único para Bases de Datos Orientadas a Objetos. Tesis Doctoral, DSIC-UPV, 1996.

[4] 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.

[5] Kifer, M. Deductive and Object Data Languages: A Quest for Integration. Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, 1995.

[6] Costa J.F., Sernadas A., Sernadas C., Ehrich H.-D. Object Interaction. INES, Lisboa, 1992.

[7] Dignum,F., Meyer J.-J.Ch., Wieringa R.J., Kuiper R.A. Modal Approach to Intentions, Commitments and Obligations: Intention plus Commitment yields Obligation, DEON'96 Workshop on deontic logic in computer science, Lisboa, 1996.

[8] 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.

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

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

[11] 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.

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

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

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

[15] Hoare, C.A.R. Communicating Sequential Processes. Prentice Hall, 1985.

[16] 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.

[17] Letelier P., Sánchez P., Ramos I. Animación de Modelos Conceptuales para ayudar en la Validación de Requisitos. ASOO'97, Buenos Aires, Argentina, 1997.

[18] Letelier P., Sánchez P., Ramos I., Pastor O. Formalización de OASIS en Lógica Dinámica incluyendo especificaciones de proceso. Informe técnico DSIC-II/2/98, Universidad Politécnica de Valencia, 1998.

[19] 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.

[20] 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.

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

[22] Pastor O. Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo OO. Tesis Doctoral, DSIC-UPV, 1992.

[23] 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.

[24] 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). Barcelona, Spain, 1997.

[25] Pernici B. Objects with Roles. In EEE/ACM Conference on Office Information Systems, Cambridge, Mass., 1990.

[26] Ramos I., Pastor O., Cuevas J., Devesa J. Objects as Observable Processes. Actas del 3rd. Workshop on the Deductive Approach to Information System Design, Cataluña, 1993.

[27] Sánchez P., Letelier P., Ramos I. Constructs for prototyping information systems with object Petri Nets. IEEE SMC'97, pages 4260-4265, Orlando, 1997.

[28] Sernadas A., Fiadeiro J., Sernadas C., Ehrich H.-D. The Basic Building Blocks of Information Systems. Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa Codex, Portugal, 1989.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

Apéndice A

Glosario

Acción. Acontecer atómico e instantáneo en la vida de un objeto. Una acción tiene tres elementos: cliente, servidor y servicio. Para el cliente una acción corresponde a un servicio que se está requiriendo a cierto servidor. Para el servidor, una acción es un servicio que está solicitando cierto cliente.

Action sending. Mecanismo de comunicación que establece que la ocurrencia de la acción de solicitar un servicio, desde un cierto cliente, tiene una correspondiente ocurrencia de acción por proveer o rechazar dicho servicio en el objeto servidor.

Action waiting. Mecanismo de comunicación que establece la espera de una cierta acción, en un objeto servidor, hasta que en un objeto cliente ocurra dicha acción.

Action calling. Mecanismo de comunicación que establece que la ocurrencia de cierta acción en un objeto implica la ocurrencia en una acción en otro objeto.

Action sharing. Mecanismo de comunicación que establece que una acción ocurre si y sólo si ocurre en todos los objetos que comparten dicha acción. La acción compartida debe corresponder a un servicio en cada uno de los objetos involucrados.

Agregación. Operador entre clases que permite modelar la noción de objeto compuesto (llamado agregado) de otros, llamados componentes.

Atributo. Propiedad estructural de los objetos de una clase. Un atributo puede ser constante o bien verse afectado por la ocurrencia de acciones (atributo variable). Además, se dice que un atributo es derivado si está defitido en función de otros atributos.

Ciclo de vida del objeto. Secuencia de pasos que le acontecen a un objeto, desde su creación hasta su posible destrucción.

Clase. Terna que contiene: un conjunto de instancias, un mecanismo de identificación para ellas y una plantilla común a todas las instancias de la clase.

Cliente. Objeto que actúa como generador de una determinada acción. El cliente de una acción es el objeto que requiere dicha acción.

Concurrencia inter-objetual. Concurrencia entre objetos del sistema, cada uno de los cuales tiene su propio hilo de ejecución.

Concurrencia intra-objetual. Concurrencia interna a cada objeto y según la cual más de una acción puede acontecerle a un objeto en un mismo instante.

Conjunto existencia de una clase. Conjunto de todas las instancias que tiene una clase en un determinado instante.

Constante agente. Nombre que permite identificar a un proceso o a los subprocesos que lo definen.

Derivación (derivation). Fórmula que establece el valor de un atributo en función de los valores de otros atributos.

Disparo (trigger). Fórmula que establece un *action sending* o *action waiting* entre un objeto cliente y un objeto servidor cuando se satisface cierta condición. Dicho de otro modo, representa el concepto de obligación asociado al estado.

Dominio. *Sort* para valores de atributos.

Especialización. Forma de herencia en la cual se definen subclases a partir de una superclase. La relación definida entre las subclases y superclase tiene la semántica *is_ a*.

Especie. Subclases de más bajo nivel en una jerarquía de especialización. Se obtienen automáticamente realizando las combinaciones posibles entre subclases de distintas particiones.

Especificación de proceso. Representación de secuencias de acciones que son permitidas (u obligadas) en la vida de un objeto.

Estado del objeto. Conjunto de fórmulas que se satisfacen en un determinado mundo del objeto.

Estado del proceso. Clase de equivalencia de estados del objeto. El estado de cada proceso del objeto está determinado por un atributo implícito.

Evaluación (valuation). Fórmula que establece una aserción en el estado alcanzado a partir de otra del estado anterior y la ocurrencia de una acción.

Evento. Servicio atómico ofrecido por un objeto y que no posee duración.

Evento compartido. Evento que debe ocurrir en más de un objeto a la vez haciendo uso del concepto de *event sharing*.

Evento privado. Evento que acontece de forma particular en la vida de un sólo objeto sin estar supeditado al acontecer en otros.

Evento destroy. Evento último en la vida de un objeto que implica su destrucción.

Evento new. Primer evento en la vida de un objeto que implica su creación.

Extensión de una clase. Conjunto de todos los posibles objetos que pueden pertenecer a una clase.

Fórmula bien formada. Fórmula definida constructivamente y utilizada para realizar aserciones respecto del estado de un objeto.

Generalización. Forma de herencia por la que una clase es definida como superclase de un conjunto de subclases. La relación definida entre las subclases y superclase tiene la semántica *is_a*.

Grupo de rol. Conjunto de clases de rol cuyos elementos son excluyentes entre sí y en los que cada uno lleva asociada una cardinalidad de simultaneidad mínima y máxima.

Herencia. Mecanismo por el cual una clase (llamada subclase) hereda propiedades (estructura y comportamiento) de otra clase (llamada superclase).

Herencia múltiple. Una clase (subclase) hereda propiedades de más de una clase (superclases). Las especies definidas basan su definición en este concepto.

Ingeniería de requisitos. Área de la ingeniería de software que estudia la captación, representación, análisis y validación de requisitos de sistemas de información.

Instancia. Objeto que pertenece a una clase.

Intensión de una clase. Conjunto de propiedades que son compartidas por las instancias de una clase.

Interfaz. Mecanismo que permite establecer propiedades de accesibilidad entre clientes y servidores.

Interfaz de proyección. Interfaz que restringe los atributos y/o servicios ofrecidos por ciertos servidores a ciertos clientes.

Mecanismo de identificación. Conjunto de pares atributo-valor que determinan un objeto de forma unívoca dentro de la población de una clase.

Modelo conceptual. Representación abstracta de los requisitos funcionales de un sistema de información.

Mundo. Estructura sobre la que se interpretan las fórmulas que caracterizan el estado del objeto. La transición válida entre mundos posibles es la base de una Estructura de Kripke.

Objeto. Unidad de modelado en el enfoque orientado a objeto, encapsulando estructura y comportamiento.

Obligación. Operador deóntico que fuerza la ocurrencia de una acción.

Oid (object identifier). Identificador implícito de cada objeto y determinado por el sistema.

Operación (operation). Secuencia de acciones que obligatoriamente deben ocurrir en la vida de un objeto. Si la operación es de tipo transacción, entonces tiene la semántica de "todo o nada" respecto del efecto de las acciones sobre el estado del objeto.

Operadores de clase. Mecanismo para especificar clases complejas. Los operadores utilizados son agregación y herencia.

Partición dinámica. Mecanismo de herencia por el que un objeto puede en un instante especializarse como instancia de alguna de las subclases definidas en dicha partición. Podrá migrar entre ellas por la ocu7rrencia de eventos o en función de los valores de sus atributos.

Partición estática. Mecanismo de herencia por el que un objeto, desde su creación, se especializa como instancia de alguna de las subclases de la partición. Su pertenencia a dicha parte sólo cesa cuando el objeto es destruido.

Paso. Conjunto de acciones que un objeto lleva a cabo en cierto instante y que le llevan a un nuevo estado.

Permiso. Operador deóntico que posibilita la ocurrencia de una acción.

Perspectiva cliente de un objeto. Punto de vista desde el cual un objeto requiere servicios.

Perspectiva servidor de un objeto. Punto de vista desde el cual el objeto provee servicios.

Plantilla. Patrón de estructura y comportamiento para un conjunto potencial de objetos.

Player. Objeto que ante la ocurrencia de una acción concreta comienza a desempeñar un papel concreto (el determinado por un objeto rol) y cuya finalización del papel viene por la destrucción del objeto que representa el rol.

Precondición (precondition). Condición que debe satisfacer el estado de un objeto para que una acción ocurra y produzca una transición entre estados.

Proceso. Desde el punto de vista del concepto de objeto, un proceso es una unidad de ejecución que puede posiblemente terminar y en cuya ejecución puede necesitar comunicarse o colaborar con otros procesos. Un objeto es entonces un proceso observable. Desde una perspectiva de vidas posibles, un proceso es una secuencia válida de acciones que le pueden ocurrir a un objeto a lo largo de su existencia.

Procesos anidados. Relación jerárquica entre procesos. Permite que una transición entre estados para un proceso en lugar de estar determinada por la ocurrencia una acción este definida por la realización de otro proceso.

Prohibición. Operador deóntico que impide la ocurrencia de una acción.

Propiedad emergente. Propiedad de una clase compleja que es adicional a las propiedades capturadas por defecto según el operador de clases complejas utilizado.

Protocolo (protocol). Secuencia permitida para la ocurrencia de un conjunto de acciones en la vida de un objeto.

Restricción de integridad (integrity constraint). Invariante de estado que deben cumplir todos los objetos que pertenecen a la clase en la que dicha restricción ha sido definida.

Rol. Estructura y comportamiento que puede desempeñar temporalmente un objeto, al cual llamamos *player*. Es importante distinguirlo de la herencia por particiones ya que un rol es un nuevo objeto con un *oid* propio.

Servicio. Evento u operación que un objeto provee.

Servidor. Objeto que provee un determinado servicio requerido mediante una acción. Objeto que recibe la petición de llevar a cabo una acción.

Subclase. Clase derivada a partir de otra mediante herencia.

Superclase. Clase a partir de la cual otras se derivan por herencia.

Transacción. Secuencia de acciones que llevan al objeto desde un estado a otro. En el estado alcanzado deben cumplirse las restricciones de integridad, de lo contrario el estado alcanzado será el mismo estado de partida.

Transición válida. La transición de un objeto desde un estado a otro válido por el acontecer de un paso.

Traza. Secuencia finita de pasos en la vida de un objeto conteniendo en el paso inicial la acción de creación de sí mismo.

Apéndice B

Sintaxis de OASIS 3.0

En este anexo se presenta la BNF completa del lenguaje de especificación asociado al modelo de OASIS. La notación utilizada es:

- **símbolo terminal** o si se trata de caracteres entre ' '
- <**símbolo no-terminal**>
- símbolos opcionales, están entre [ ]
- símbolo alternativos, están separados por |

Al exponer la sintaxis de un determinado elemento del lenguaje se usarán las siguientes simplificaciones: siendo x un elemento en particular, para describir un bloque de elementos x usaremos <bloque_x>; para una lista de elementos x usaremos <lista_x> y para una secuencia de elementos x usaremos <sec_x>. Los significados asociados a cada uno de ellos se establecen a continuación.

<bloque_x>	::=	<bloque_x> <x> | <x>
<lista_x>	::=	<lista_x> ',' <x> | <x>
<sec_x>	::=	<sec_x> <x> ';'| <x> ';'

Además, para cada *elemento* que tenga asociado un nombre como identificador se usará el no terminal <id_*elemento*>, que se corresponde con una cadena de caracteres.

B.1. Esquema conceptual

<def_esquema_conceptual>	**::=**	**conceptual schema** <id_esquema> [<bloque_def_dominio>] <bloque_def_clase> [<bloque_def_clase_compleja>] <bloque_def_interfaz> **end conceptual schema**

B.2. Definición de dominio

<def_dominio>	**::=**	**domain** <id_dominio> '**(**' <lista_id_dominio> '**)**' [**import** <lista_id_dominio>] '**;**' [**var** <lista_def_variable>] '**;**' **functions** <sec_def_función> **equations** <sec_def_ecuación> **end domain**
<def_variable>	**::=**	<id_variable> '**:**' <dominio>
<def_función>	**::=**	<id_función> ['**(**' lista_def_parámetro '**)**'] '**:**' <dominio>
<def_ecuación>	**::=**	<fórmula_ecuacional>[1]

[1]Fórmula bien formada de la lógica ecuacional, del tipo $ter_1 = ter_2$, donde ter_1 y ter_2 son términos que describen cómo trabajan las funciones.

B.3. Clases

<def_clase>	::=	**class** <id_clase> <mecanismos_identificación> <atributos_constantes> [<atributos_variables>] [<atributos_derivados>] [<derivaciones>] [<restricciones_integridad>] <eventos> [<operaciones>] [<disparos>] [<evaluaciones>] [<precondiciones>] [<protocolos>] **end class**

B.3.1. Mecanismos de identificación

<mecanismos_identificación>	::=	**identification** <sec_def_identificador>
<def_identificador>	::=	<id_identificador> ':' '**(**'<lista_id_atributo>'**)**'

B.3.2. Atributos constantes y variables

<atributos_constantes>	::=	**constant attributes** <sec_def_atributo_cons_o_var>
<atributos_variables>	::=	**variable attributes** <sec_def_atributo_cons_o_var>
<def_atributo_cons_o_var>	::=	<id_atributo>'**:**' <dominio> ['**(**'<lista_valor_por_defecto>'**)**'] [**towards** <cardinalidad> [<def_representación>]]
<valor_por_defecto>	::=	<valor>

B.3.3. Atributos derivados

<atributos_derivados>	::=	**derived attributes** <sec_dec_atributo_derivado>
<dec_atributo_derivado>	::=	<id_atributo> '**:**' <dominio>

B.3.4. Derivaciones

<derivaciones>	::=	**derivations**
		<sec_def_atributo_derivado>
<def_atributo_derivado>	::=	['**{**'<fórmula>'**}**'] <asignación>
<asignación>	::=	<id_atributo> '**:=**' <expresión>

B.3.5. Restricciones de integridad

<restricciones_integridad>	::=	**constraints** <sec_def_restricción>
<def_restricción> ::=		
		[**always**] '**{**'<fórmula>'**}**'
		| **sometimes** '**{**'<fórmula>'**}**'
		| **next** '**{**'<fórmula>'**}**'
		| <fórmula_después> **since** <fórmula_antes>
		| <fórmula_antes> **until** <fórmula_después>
		| **sometimes** ['**(**'<timeout>'**)**'] <fórmula_después>
		since <fórmula_antes>
		| **always** ['**(**'<delay>'**)**'] <fórmula_después>
		since <fórmula_antes>
<fórmula_antes>	::=	'**{**'<fórmula>'**}**'
<fórmula_después>	::=	'**{**'<fórmula>'**}**'
<timeout>	::=	<op_rel_timeout> <natural>
		<unidad_tiempo>
<delay>	::=	<op_rel_delay> <natural>
		<unidad_tiempo>
<op_rel_timeout>	::=	'<' | '<='
<op_rel_delay>	::=	'>' | '>='
<unidad_tiempo>	::=	**seconds** | **minutes**| **hours** | **days**
		| **weeks** | **months** | **years**

B.3.6. Eventos

<eventos>	::=	**events**
		<sec_def_evento>
<def_evento>	::=	
		<id_evento> ['**(**' lista_def_parámetro '**)**']
		[**new** | **destroy**]
		[**alias for** <operación_implícita>[2]| <coordinación>]
<coordinación>	::=	**calling to members**
		<lista_servicio_componente>
		| **sharing with members**
		<lista_servicio_componente>

B.3.7. Evaluaciones

<evaluaciones>	::=	**valuations**
		<sec_def_evaluación>
<def_evaluación>	::=	['**{**'<fórmula>'**}**'] '**[**' <acción> '**]**'
		<lista_asignación>
<asignación>	::=	<id_atributo> '**:=**' <expresión>

B.3.8. Precondiciones

<precondiciones>	::=	**preconditions**
		<sec_def_precondición>
<def_precondición>	::=	<acción> **if** '**{**'<fórmula>'**}**'

B.3.9. Disparos

<disparos>	::=	**triggers**
		<sec_def_disparo>
<def_disparo>	::=	<acción> **when** '**{**'<fórmula>'**}**'

[2]Operación implícita aplicable a atributos o componentes multivaluados, tal como se definen en el apartado de clases complejas dentro del capítulo 6.

B.3.10. Protocolos y operaciones

<operaciones>	**::=**	**operations** <bloque_def_operación>
<def_operación>	**::=**	<id_operación> ['**(**'lista_def_parámetro'**)**'] [**transaction**] '**:**'<sec_def_estado>
<protocolos>	**::=**	**protocols** <bloque_def_protocolo>
<def_protocolo>	**::=**	<id_protocolo> ['**(**'lista_def_parámetro'**)**'] '**:**' <sec_def_estado>

B.4. Clases complejas

< def_clase_compleja >	::=	< def_agregación > | < def_herencia >

B.4.1. Agregación

<def_agregación>	**::=**	<id_clase> **aggregation of** <partes>
<partes>	**::=**	<sec_def_componente> | <def_asociación>
<def_componente>	**::=**	[**static** | **dynamic**] [**inclusive** | **relational**] [<id_alias> **alias for**] <id_clase_componente> [**towards** <cardinalidad> [<def_representación>]] [**from** <cardinalidad> [<def_representación>]]
<def_asociación>	**::=**	<id_class_componente> **grouping by** <lista_id_atributo> [**where '{'**<fórmula>'**}**'] [**towards** <cardinalidad> [<def_representación>]] [**from** <cardinalidad> [<def_representación>]]

B.4.2. Herencia

<def_herencia>	::=	<def_partición_estática> | <def_partición_dinámica> | <def_roles>

Partición estática

<def_partición_estática>	::=	<lista_id_subclase> **static specialization of** <id_clase>

Partición dinámica

<def_partición_dinámica>	::=	<def_dinámica_migración> | <def_dinámica_atributo>

Partición dinámica por expresión de migración

<def_dinámica_migración>	::=	<lista_id_subclase> **dynamic specialization of** <id_superclase> **migration relation is** <expresión_migración>
<expresión_migración>	::=	<sec_def_estado>

Partición dinnámica en función de valores de atributos

<def_dinámica_atributo>	::=	<lista_def_subclase_dinámica> **dynamic specialization of** <id_superclase>
<def_subclase_dinámica>	::=	<id_subclase> **where** '**{**'<fórmula>'**}**'

Roles

<def_roles>	::=	<lista_def_rol> **role of** <id_clase_player>
<def_rol>	::=	<id_clase_rol> [**towards**'**(**'<card_min>'**,**'<card_max>'**)**'] <id_evento_rol>

B.5. Interfaces

<def_interfaz>	::=	**interface** <ref_cliente> **with** <ref_servidor> [**attributes** '**(**' <atributos_ofrecidos> '**)**'] **services** '(' <servicios_ofrecidos> '**)**' **end interface**
<ref_cliente>	::=	<ref_objeto>
<ref_servidor>	::=	<ref_objeto>
<atributos_ofrecidos>	::=	**all** | **none** | <lista_id_atributo>
<servicios_ofrecidos>	::=	**all** | **none** | <lista_id_servicio>

B.6. Elementos comunes

B.6.1. Definición de estados de proceso

<def_estado>	::=	<id_estado>'**=**'<términos_estado>
<términos_estado>	::=	<un_término_estado> | <términos_estado> '**+**' <términos_estado>
<un_término_estado>	::=	<término_con_guarda> | <término_sin_guarda>
<término_con_guarda>	::=	'**{**'<fórmula>'**}**'<término_sin_guarda>
<término_sin_guarda>	::=	<acción_proceso> ['**.**' <id_estado>] | <proceso> ['**.**' <id_estado>]
<acción_proceso>	::=	[<cliente_proceso>] [<servidor_proceso>] <servicio>
<cliente_proceso>	::=	<cliente> | **client** '**:**'
<servidor_proceso>	::=	<servidor> | **server** '**::**'
<proceso>	::=	<id_operación> ['**(**' lista_parámetro '**)**'] | <id_protocolo> ['**(**' lista_parámetro '**)**']

B.6.2. Acción

<acción>	::=	[<cliente>] [<servidor>] <servicio>
<cliente>	::=	<ref_objeto> '**:**'
<servidor>	::=	<ref_objeto> '**::**' | '**::**'
<servicio>	::=	<id_evento> | <id_operación> | <servicio_componente> | <operación_implícita>

B.6.3. Referencia a objeto

<ref_objeto>	**::=**	**self** | <especie> [<identificacion>]
<especie>	**::=**	<id_clase> | <id_clase> '*****' <especie>
<identificación>	**::=**	'(**someone**)'
		| '(**everyone**)'
		| '**(**' <id_identificador> '**,**'
		'**[**'<lista_valor_atributo> '**]**' '**)**'
<valor_atributo>	**::=**	<expresión> | '**_**'[3]

B.6.4. Fórmulas

<fórmula>	**::=**	
		'**(**' <fórmula> '**)**'
		| **not**'**(**' <fórmula> '**)**'
		| <fórmula> **and** <fórmula>
		| <fórmula> **or** <fórmula>
		| <expresión_aritmética> <op_rel> <expresión_aritmética>
		| **true** | **false**
<op_rel>	**::=**	'=' | '<>' | '<' | '<=' | '>' | '>='

B.6.5. Atributos y servicios de componentes

<atributo_componente>	**::=**	<ref_objeto>'.' <id_atributo>
		| <ref_objeto>'.'<atributo_componente>
<servicio_componente>	**::=**	<ref_objeto>'.' <servicio>
		| <ref_objeto>'.' <servicio_componente>

[3]Este símbolo en la posición del valor de un atributo representa a "cualquier valor". Así, es posible hacer referencia a más de un objeto.

B.6.6. Expresiones

<expresión>	::=	<expresión_aritmética> | '**{**'<fórmula>'**}**'
<expresión_aritmética>	::=	
		'**(**' <expresión_aritmética> '**)**'
		| <operando>
		| <operando> <operador> <expresión_aritmética>
<operando>	::=	<valor> | <atributo>
		| <atributo_componente> | <parámetro>
		| <función>[4]
<operador>	::=	'+' | '-' | '*' | '/'
<valor>[5]	::=	<natural> | <entero> | <real>
		| <bool> | <caracter> | <letra>
		| <cadena> | <fecha> | <time> | <blob>

B.6.7. Dominios

<dominio>	::=	<id_dominio> | **nat** | **int** | **real** | **bool**
		| **char** | **string** | **date** | **time** | **blob**

B.6.8. Parámetros

<def_parámetro>	::=	<id_parámetro> '**:**' <dominio>

B.6.9. Cardinalidad

<cardinalidad>	::=	'**(**' <card_min> '**,**' <card_max> '**)**'
<card_min>	::=	<natural>
<card_max>[6]	::=	<natural> | '*****'

[4] Función predefinida para el dominio correspondiente o función implícita para los atributos o componentes multivaluados, tal como se presentó en el apartado de clases complejas en el capítulo 6.

[5] Corresponden a constantes en cada uno de los dominios predefinidos.

[6] El terminal '*****' denota que no existe restricción a la cardinalidad máxima.

B.6.10. Representación de atributo o componente multivaluado

<def_representación>	**::=**	**list** '**[**' <def_índice> '**]**' | **set** | **bag**
<def_índice>	**::=**	<rango> | <lista_cadena>
<rango>	**::=**	<min_rango> '**...**' <max_rango>
<min_rango>	**::=**	<natural> | <letra>
<max_rango>[7]	**::=**	<natural> | <letra> | '*****'

[7]El terminal '*****' denota que no existe restricción al límite máximo del rango.

Índice alfabético