Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos

Tesis Doctoral presentada por Oscar Pastor López
Dirigida por Isidro Ramos Salavert

Valencia, 4 de Abril de 1992
D.S.I.C. - U.P.V.

A Carmen, Regina y Oscar

No quisiera en estos momentos dejar de agradecer a Hewlett-Packard (HP) su contribución al desarrollo de esta Tesis, traducida en dos becas de investigación de dos meses cada una en los Laboratorios de Investigación de HP sitos en Bristol (Inglaterra).

Esas estancias han supuesto avances significativos en la ejecución del trabajo, y han generado un productivo trabajo en equipo con el departamento de Ingeniería de Software de dichos laboratorios.

También quiero agradecer a todos los compañeros del Grupo de Investigación todas las sugerencias e intercambios de impresiones que han enriquecido el contenido de esta memoria.

Finalmente, quiero dejar constancia de mi reconocimiento al contínuo trabajo de Isidro Ramos como Director de esta Tesis. En él siempre he encontrado apoyo, soluciones a problemas y valiosos consejos y directrices.

Valencia, 4 de Abril de 1992

Indice

Capítulo 1

Introducción

1.1 Motivación

Las metodologías de Ingeniería de Software utilizadas habitualmente han apostado por el uso de un modelo en cascada como paradigma de ciclo de vida del software. Aunque es evidente que algunas de estas metodologías (por ejemplo, Yourdon-Constantine, Warnier, metodologías de diseño de Jackson,...) están muy extendidas, las críticas que reciben son cada vez mayores ([3,7,14]). Nuevas metodologías, como prototipación automática ([2,123]), ejecutabilidad de especificaciones ([140]), programación automática y transformacional ([7]), han aparecido con el objetivo de mitigar las carencias asociadas al uso del ciclo de vida clásico. Estas nuevas metodologías comparten la idea de que un nuevo paradigma es necesario en el area de la ingeniería de software, idea que constituye el punto de partida de este trabajo.

Esta idea se plasma en la constatación que la producción de Software con una calidad industrial implica profundos cambios en la tecnología asociada. Las aproximaciones clásicas de desarrollo de Sistemas de Información (SI) basadas en los modelos en cascada no han conseguido mitigar el problema de los altos costes y baja calidad asociados a la producción de Software fiable.

El incremento de la productividad y de la calidad sólo son posibles mediante la automatización total o parcial del proceso productivo (ciclo de vida) y el uso de herramientas conceptuales, metodológicas y de producción de carácter formal que den sentido a los procesos de validación (mediante la prototipación automática) y de auditoría. Asimismo, se hace necesario que términos como consistencia, completitud etc. recuperen su significado y utilidad en la Ingeniería del Software, actualmente perdido en la jerga del CASE tradicional.

Un intenso trabajo de investigación viene desarrollándose durante los últimos años con el objetivo de profundizar en la utilización de técnicas formales de especificación basadas en diferentes lógicas (relacional o clausal, funcional o ecuacional, objetual, modal), sus fusiones y extensiones, con el fin de proporcionar una capacidad expresiva lo más potente posible preservando una semántica declarativa bien definida y una semántica operacional eficiente.

Con ello, la validación por prototipación automática del Software a desarrollar está garantizada permitiendo además abordar los citados

problemas clásicos de consistencia, completitud,... en un marco verdaderamente formal.

Es muy elevado el número de Grupos de Trabajo y Publicaciones en Congresos dedicados al estudio intensivo de los fundamentos teóricos y metodológicos del diseño de Sistemas de Información, con el objetivo último de diseñar y desarrollar Sistemas fiables, correctos y con altos niveles de reusabilidad. Todo ello especialmente a través del uso de técnicas adecuadas y formales de diseño orientado a objeto.

¿Qué aporta realmente la utilización de un Paradigma de Programación Automática como eje de un nuevo Ciclo de Vida para la producción de Software? ¿Por qué esa moderna vorágine investigadora centrada en el uso de métodos formales? ¿Qué justifica en ese contexto la ascensión del Modelo **Orientado a Objetos** (OO) más allá del mero campo de los Lenguajes de Programación?

Estas no son ya afortunadamente preguntas sin respuesta. Es un hecho aceptado que la utilización de un Ciclo de Vida en Cascada se ha convertido en los últimos años en una especie de estándar industrial para la producción de Software.

Pero los problemas derivados del uso estricto de tales Ciclos de Vida en Cascada son bien conocidos y han dado origen a un concepto ampliamente extendido en la actualidad: la 'Crisis del Software'. Entre estos problemas podemos destacar:

* Se rebasa la estimación inicial de costes.
* Retrasos en la entrega del producto final.
* Rendimientos inadecuados.
* Modificaciones muy costosas.
* Falta de fiabilidad del producto final.

Este último aspecto es probablemente el más crítico, y pone de relieve que el 'gap' semántico existente entre qué es el SI realmente y cómo es representado en un ordenador es demasiado grande.

El uso de técnicas estructuradas aplicadas al Análisis y Diseño no ha supuesto una solución definitiva como se pensaba en la década de los ochenta. Aunque la amplia difusión de las herramientas CASE ha

hecho más asequible el desarrollo de Sistemas, los problemas de base no han desaparecido porque residen en la esencia del propio paradigma utilizado.

Básicamente, cuatro son las razones esenciales que explican los problemas clásicos en el proceso de producción de Software:

1) escaso esfuerzo en las fases de Análisis y Diseño.

2) uso de lenguajes informales.

3) uso de modelos inadecuados para describir los SI reales.

4) compleja interacción hombre-máquina.

La resolución de estos problemas que parecen hasta hoy endémicos cuando queremos desarrollar Software de calidad, pasa por un cambio radical de actitud frente al desarrollo de entornos de producción de Software. Tal cambio de actitud se traduce respectivamente en:

1) Modificar el Ciclo Clásico en Cascada introduciendo Prototipación dentro de un Paradigma de Programación Automática.

2) Utilizar Lenguajes de Especificación Formales.

3) Uso de un Modelo Orientado a Objetos.

4) Desarrollar Entornos de Trabajo Avanzados Gráficos.

Es así como respondemos a las preguntas planteadas inicialmente. En tales entornos de trabajo, el Paradigma de Programación Automática se convierte en una alternativa real al Ciclo Clásico de Vida, fundamentada en el uso de métodos formales para desarrollar Software. Si disponemos de Lenguajes de Especificación Formales con suficiente potencia expresiva para describir SI, podemos obtener una especificación formal del SI considerado.

Y más aún. Si el lenguaje de especificación posee propiedades semánticas bien caracterizadas, las especificaciones podrán ser ejecutadas, comprobando automáticamente su corrección (y otras propiedades lógicas). Esto se consigue generando los programas lógicos que sean

equivalentes a la especificación. De esta forma estaremos trabajando realmente en entornos con Prototipación Automática.

Por otra parte, la utilización de aproximaciones basadas en el Modelo Orientado a Objetos está cada vez más extendida. El paradigma objetual resulta muy útil para abordar las etapas clásicas de Análisis, Diseño e Implementación. Son numerosas las Metodologías existentes de Análisis Orientado a Objeto (AOO, [28]) y Diseño Orientado a Objeto (DOO, [15]), además de una amplia y conocida colección de Lenguajes de Programación Orientados a Objeto (C++ ([121]), Smalltalk ([56]), Eiffel ([84]) etc.).

Si los Lenguajes de Especificación formales que usamos proporcionan una expresividad OO, estaremos aumentando considerablemente su capacidad expresiva debido a que los conceptos OO permiten modelar fenómenos del mundo real de una forma natural e intuitiva. Con las aproximaciones OO, el 'gap' semántico que comentábamos antes se reduce porque la descripción que se hace del SI se centra directamente en los objetos que componen el Sistema.

Como Constantine argumenta en [33], en los métodos estructurados convencionales el modelo de un problema o aplicación y el modelo del Software que resuelve el problema son notablemente distintos y se representan utilizando notaciones totalmente diferentes. Por el contrario, introduciendo una organización OO se hace posible el diseño de Software que modeliza la estructura de una aplicación de una forma muy próxima a cómo se percibe ésta en la realidad. Al menos en principio se puede afirmar que los modelos desarrollados con un AOO y un DOO pueden expresarse en una notación equivalente basada en unos principios comunes.

Esta es la principal razón que justifica el uso del modelo OO en esa nueva Metodología de Producción de Software que aquí se reivindica y que se va a desarrollar a lo largo de este trabajo. El uso de tal aproximación va a permitir modelar un Sistema como una Sociedad de Objetos, y mantener la noción de objeto durante todo el Ciclo de Vida de Desarrollo de Software. En particular, las fases de Análisis y Diseño se convierten en un proceso incremental que va refinando la imagen OO que tenemos del SI desarrollado.

1.2 Estado del arte

Como hemos señalado en el punto anterior, es mucho el trabajo de investigación que se viene desarrollando en el campo del Diseño de SI con el objetivo de profundizar en sus fundamentos teóricos y metodológicos. Particularmente interesante es la confluencia de intereses con respecto a los trabajos del grupo IS-CORE[1] . Este grupo centra sus actividades en el uso de la Lógica, el Algebra y la Teoría de Categorías para dotar de una semántica precisa a un amplio espectro de Lenguajes y Metodologías que soportan un Diseño de SI modular, activo y OO.

El hecho de apostar por la utilización de métodos formales a partir de un modelo OO y dentro de un entorno de producción de Software fiel al paradigma de Programación Automática, concentra el trabajo a desarrollar en las siguientes Líneas de Investigación:

1) Semántica del Modelo OO.

2) Lenguajes OO para el Diseño de SI.

3) Uso de Lógicas Clausales y Ecuacionales en una Especificación OO.

4) Métodos AOO y DOO que conformen el núcleo de una Metodología de Producción Automática de Software.

1.2.1 Semántica del Modelo OO

Es obvio que el nivel de desarrollo práctico alcanzado por los métodos OO en los campos más relevantes de la Informática es muy alto. Empezando por los trabajos en Lenguajes de Programación Orientados a Objeto (LPOO) [56,84,121], y continuando con Bases de Datos Orientadas a Objetos (BDOO) [6,74,47,73] y Lenguajes de Especificación de alto nivel [23,70,95,115], una gran cantidad de trabajo ha sido desarrollado.

[1] El grupo IS-CORE (Information Systems-COrrectness and REusability) participa en el proyecto ESPRIT-2 BRA, y forman parte de él como responsables A.Sernadas (INESC (Lisboa)), H.-D.Ehrich (Technical University of Braunschweig (Alemania)), U.Lipeck (University of Hannover (Alemania)), T.S.E.Maibaum (Imperial College (Londres)) y R.Meersman (Tilburg University (Holanda))

Pero es curioso constatar que el establecimiento de los fundamentos teóricos de los conceptos básicos OO no ha evolucionado paralelamente. Sólo recientemente tales enfoques formales han empezado a aparecer, pero es indudable que los aspectos prácticos van todavía muy por delante.

Si se quiere trabajar en un entorno basado en un modelo OO, obviamente es una cuestión básica el fijar con precisión qué es lo que quiere decir OO. Los trabajos de formalización que vienen realizándose pueden agruparse en dos grandes grupos:

1) Aproximaciones algebraicas y categóricas.

2) Aproximaciones ontológicas.

Aproximaciones algebraicas y categóricas

Los trabajos realizados en el area de especificación de Tipos Abstractos de Datos (TAD) han proporcionado la base necesaria para formalizar entornos de especificación complejos [55]. Tradicionalmente, el enfoque TAD se centra en torno a valores, en lugar de a objetos, por lo que algunos investigadores lo consideran inapropiado para formalizar un modelo OO [46].

En cualquier caso, una formalización del modelo OO debe interpretar un Sistema como una sociedad de objetos interactivos. Dos tareas son en este contexto necesarias:

1) una noción precisa de objeto

2) un modelo para la interacción entre objetos

Los trabajos de Sernadas y Ehrich presentando OBLOG en el contexto de una Teoría de Tipos Abstractos de Objetos [43,111,116] proporcionan un sólido conjunto de resultados teóricos y una propuesta formal de definición del modelo OO usando la Teoría de Categorías. Brevemente, un objeto es formalizado como una 4-tupla compuesta por un conjunto de atributos, un conjunto de eventos, un conjunto de trazas o ciclos de vida (compuesto por secuencias admisibles de eventos) y una función de observación, y se introduce la noción de morfismo entre objetos para

dar cuenta de su interacción, caracterizando de esta forma una sociedad de objetos como una Categoría.

La característica esencial de esta aproximación es que los objetos son vistos como procesos con atributos dependientes de una traza. En diversos trabajos previos [42,43,44,110] se presentan diferentes *categorías de procesos*, en los que se modela la composición paralela de procesos a través de la noción categórica de *límite.* Los límites son construcciones muy potentes en Teoría de Categorías, y su utilidad para describir comportamiento es conocida desde hace tiempo ([51,52,53]).

En todos estos trabajos se trata de separar la modelización de la parte estructural (datos) de los objetos y la de la parte dinámica (procesos). En [41] se realza la uniformidad estructural de dichas partes y se caracterizan ambas como instancias de un concepto más general llamado *comportamiento.*

Partiendo de la definición de un objeto como una 4-tupla, MOL [23,24,91,93,94] asciende al nivel de clase y adopta una aproximación puramente algebraica, definiendo una clase como una estructura algebraica cuyos principales componentes son un conjunto de eventos, un conjunto de atributos, un conjunto de trazas que representan secuencias admisibles de eventos y que dan cuenta de la vida de un objeto, y una función de observación que permite conocer el estado de un objeto en función de sus eventos relevantes.

En este contexto, las clases complejas se definen a partir de clases elementales utilizando un conjunto de operadores entre clases con los que dar cuenta de las relaciones estructurales existentes entre las clases del SI estudiado.

El uso de un modelo OO posibilita la encapsulación de aspectos estáticos y dinámicos bajo la noción de objeto. Ello proporciona un marco de referencia común en el que incorporar propiedades estructurales conocidas en el area de la Modelización Conceptual, y características dinámicas propias de estudios preocupados por especificación de comportamiento como es el caso de las álgebras de procesos. En esta línea, la introducción como operadores entre clases de los mecanismos de abstracción clásicos en el mundo de la Modelización Semántica (agregación, especialización y asociación) más otro operador muy utilizado en el campo del álgebra de procesos (composición paralela) proporciona un mecanismo constructivo para definir la especificación de un SI como

la clase compleja obtenida por composición paralela de todas las clases definidas previamente.

Profundizando en esta perspectiva algebraica del modelo OO, la extensión activa de MOL, OASIS, presenta una propuesta formal simple y potente que analizaremos detenidamente en esta Tesis. Esta aproximación es el punto de partida para una línea de trabajo actualmente en desarrollo, cuyo objetivo final es la formalización de un entorno de especificación OASIS en el que la noción de clase se define como un término con variables del lenguaje de términos con variables de una cierta signatura (característica del lenguaje).

Una clase así definida denota un sublenguaje de términos constantes (sin variables). Si definimos una relación de equivalencia entre tales términos constantes de forma que dos términos son equivalentes si y solo si sus atributos clave tienen el mismo valor, aparece el concepto de **tipo** (población de una clase) como el conjunto de las Clases de Equivalencia inducidas, y el de **objeto** denotando cada una de las clases de equivalencia inducidas en un tipo cuando sus términos se relacionan si poseen el mismo identificador.

Como se comenta en [68], la investigación en desarrollo continua también en distintas direcciones.

Por un lado, profundizando en las Semánticas basadas en trazas, los modelos presentados en [34,35] evolucionan progresivamente de un modelo basado en un estricto entrelazado a un modelo con concurrencia plena. Estos modelos investigan la estrecha relación que ha sido establecida entre los conceptos de objeto y proceso ([110]), por la que la semántica de la agregación de objetos es analizada en términos de la composición paralela de procesos. Los resultados de esta linea de trabajo han originado algunas contribuciones a la propia Teoría de Procesos, especialmente en lo que concierne a la Teoría de Categorías (construcción universal) como un medio válido de formalización de técnicas de composición de procesos, particularmente en presencia de interacción (compartición de eventos o llamadas) ([36]), y de formalización de iniciativa (eventos activos) y requerimientos transaccionales ([35]).

Por otra parte, dentro también de los trabajos del grupo IS-CORE y en colaboración con el Prof.J.Goguen, se han desarrollado nuevos modelos objetuales ([41]) que formalizan los objetos como procesos observados sin hacer referencia a un tiempo específico o a un dominio

espacial (lineal, ramificado, discreto, continuo,etc.). De esta forma se dispone de un marco más general donde las técnicas de composición de objetos pueden ser analizadas independientemente de cualquier modelo específico de evolución.

Aproximaciones ontológicas

Otra aproximación interesante es la Ontológica [130]. Parte de la idea de que el modelo objetual está cada vez más extendido debido a que el uso de la noción de objeto permite tener una visión natural del mundo que modelamos, pues se corresponde directamente con conceptos reales.

Como la rama de la Filosofía de la Ciencia que trata de la existencia de las cosas en el mundo es la Ontología (una parte de la Metafísica), Wand utiliza una visión ontológica para sentar la base formal de la noción de objeto. El formalismo usado tiene su origen en [20,21].

El modelo objetual presentado permite definir objetos independientemente de cualquier consideración de implementación. Tiene como principales novedades la distinción explícita entre objetos y propiedades, y el tratamiento de la dinámica de un Sistema de Objetos a través del concepto de ley.

1.2.2 Lenguajes OO para el Diseño de SI

Una vez presentada una propuesta de definición formal del modelo objetual, el paso siguiente va a ser desarrollar Lenguajes de Especificación OO que den cuenta correctamente de dicho modelo y que puedan ser utilizados como herramienta formal de investigación.

El trabajo en el área de Lenguajes se centra fundamentalmente en identificar las estructuras básicas necesarias para hacer factible una especificación OO ([72,95]). Esto incluye la descripción de clases primitivas (simples, atómicas), la descripción de clases elementales, y la de clases compuestas obtenidas haciendo uso de un cierto conjunto de operadores entre clases, del que son siempre miembros de una forma u otra los mecanismos de herencia y agregación.

Este esfuerzo en el desarrollo de lenguajes se realiza tanto en un contexto textual como visual. Entre los primeros podemos señalar

los trabajos que están siendo desarrollados principalmente en TUBS[2] ([71,106]), y que han sido dirigidos a objetos compuestos [103,71] y especificación de bases de datos y modelización conceptual [69,104,105]. El lenguaje visual está siendo desarrollado en el INESC[3][117]. El trabajo en este contexto se concentra en dar soporte a la declaración de objetos compuestos y desarrollar mecanismos para la especificación gráfica de Sistemas Complejos (compuestos por un gran número de objetos) ([114,115]). Entornos de prototipación gráfica también están siendo desarrollados en el DSIC-UPV[4] [45], incluyendo editores, traductores y ejecutadores (evaluadores) tanto textuales como gráficos.

Otra linea de investigación estudia la relación existente entre Bases de Datos OO y Deductivas. En [91] se presenta un entorno de trabajo que genera un prototipo de Base de Datos Deductiva a partir de una especificación OO, siendo el prototipo generado formalmente equivalente a la especificación. En [113,112] el objetivo primordial es transformar requerimientos de evolución de un objeto en una especificación de su comportamiento transformacional (efectos de los eventos sobre los atributos) y reactiva (restricciones y requerimientos para la ocurrencia de eventos).

Obviamente, el desarrollo de Lenguajes de Especificación OO está íntimamente ligado a la semántica asociada al modelo OO, para proporcionar a las construcciones sintácticas del Lenguaje una Semántica operacional y declarativa conforme a la definición dada.

1.2.3 Uso de Lógicas Clausales y Ecuacionales en una Especificación OO

Son muchas las propuestas de combinación de diferentes lenguajes lógicos correspondientes a distintos paradigmas de programación en un lenguaje coherente único. En esta línea [54] presenta algunos principios de diseño para lenguajes que unifiquen los paradigmas funcional, OO y relacional (clausal), dando también una semántica a tales lenguajes

[2]Technical University of Braunschweig (Alemania)

[3]Instituto Superior Técnico (Lisboa,Portugal)

[4]Departamento de Sistemas Informáticos y Computación,Universidad Politécnica de Valencia

unificados.

Si en lugar de realizar tal fusión de paradigmas a nivel de programación, se realiza a nivel de especificación tomando como base un modelo OO formalmente definido, podemos generar programas lógicos (relacionales, funcionales o generales) que sean formalmente equivalentes a la especificación de partida, adaptando a la semántica del lenguaje integrado una aproximación operacional basada en traducción.

En esto contexto se situan los entornos de Prototipación Automática FMOL [23,94] en los que el Lenguaje de Especificación utiliza una expresividad OO y funcional, y RMOL [91,93] en el que los paradigmas fusionados son el OO y el clausal. En ambos casos, un programa lógico equivalente a la especificación es generado automáticamente por un traductor. El lenguaje lógico objeto es Axis ([29]) y Rap ([66]) para FMOL, y Prolog ([13]) para RMOL.

En los últimos años se han desarrollado varias aproximaciones a la integración de la programación clausal y ecuacional, bien mediante adición de variables lógicas a un lenguaje funcional para obtener un lenguaje lógico con sintaxis funcional (la semántica operacional de este tipo de lenguajes es 'narrowing'), bien incorporando igualdad a un lenguaje lógico para obtener un lenguaje lógico con características funcionales (la semántica operacional es en este caso SLDE-resolución, paramodulación o flat-SLD-resolución).

En la medida en que tales propuestas de unificación se traduzcan en la aparición de lenguajes de programación lógico-ecuacionales eficientes, se podrá ampliar la expresividad de los lenguajes de especificación de forma que se pueda hacer uso del formalismo más adecuado para definir las clases componentes de un SI. Paralelamente se generarán los traductores que obtengan automáticamente el programa lógico general equivalente a cualquier especificación. En [40,37] se presentan entornos de estas características que utilizan como lenguaje lógico objeto Europa ([4]).

1.2.4 Métodos de AOO y DOO en el contexto de una Metodología de Producción Automática de Software

El modelo OO ha influido espectacularmente en todas las fases del Ciclo de Vida de Desarrollo de Software, incluidas las más tempranas. Las técnicas tradicionales de Análisis Estructurado, tipificadas en los trabajos de De Marco [80], Yourdon [139] y Gane y Sarson [48], con extensiones para Sistemas de tiempo real efectuadas por Ward y Mellor [133], abordan el desarrollo de SI desde dos perspectivas distintas:

1) analizar el flujo de datos característico de un Sistema para dar cuenta de sus aspectos dinámicos o de comportamiento.

2) elaborar un Modelo Semántico [65] para representar la estructura estática del Sistema.

El AOO tiene como principal objetivo la construcción de modelos del mundo real a través de una vista OO de la realidad. Entre las ventajas asociadas al uso de técnicas de AOO cabe destacar tres:

1) El modelo encapsula aspectos estáticos (datos) y dinámicos (procesos) bajo una notación común.

2) Se trata de un modelo cercano a los mecanismos cognitivos humanos, ya que consiste básicamente en abstraer nuestro conocimiento del dominio del problema a través de un proceso de clasificación.

3) Dentro de un entorno OO, la barrera clásica existente entre las fases de Análisis y Diseño se difumina, ya que en ambos casos se manipulan clases, objetos y relaciones entre objetos. De esta forma el Diseño ya no es una fase distinta y monolítica, sino que se convierte en un paso más dentro del desarrollo iterativo e incremental de un SI que las aproximaciones OO posibilitan.

El esfuerzo que se viene realizando en la elaboración de métodos de AOO y DOO es muy grande. Son ya numerosos los métodos de AOO que han sido publicados. Destacan entre ellos las propuestas de Coad y Yourdon [28], Booch [15], Rumbaugh [98] y sus extensiones [31] , Mellor y Shlaer [82], etc.

Pero la mayoría de ellos intentan adaptar metodologías tradicionales basadas en métodos estructurados. Ese intento de integración es sospechoso. Constituye un ejemplo típico de una situación en la que un enfoque que se queda anticuado inventa algo supuestamente nuevo para sobrevivir. Además, están basados en estructuras y conceptos procedentes de los Lenguajes de Programación OO (LPOO).

En cualquiera de las dos situaciones anteriores se plantean problemas inmediatamente:

1) porque el Análisis Estructurado mezcla información acerca de la comunicación entre objetos con información referente al ciclo de vida local al objeto. Además utiliza diferentes modelos para dar cuenta de aspectos estructurales y de comportamiento, lo que hace muy difícil hablar de consistencia entre modelos.

2) porque el Análisis es una forma de expresar un área compleja de la realidad que debe ser ajena a consideraciones de implementación. Por tanto, partir de las técnicas de implementación de los LPOO actuales para elaborar los métodos de especificación OO ensombrece la potencia expresiva de tales métodos.

El camino defendido en esta Tesis es el contrario: partamos de una definición formal del modelo OO, y diseñemos Lenguajes de Especificación que sean verdaderamente OO de acuerdo con la definición formal dada. Estos Lenguajes constituirán el núcleo de un entorno de producción de Software basado realmente en métodos OO que abarquen todas las fases del proceso de producción de Software (Análisis, Diseño e Implementación) permaneciendo fieles al Paradigma de Programación Automática.

En este contexto resulta especialmente útil el disponer de un criterio de clasificación que actúe como marco de referencia en el que situar diferentes propuestas de Lenguajes.

Un criterio simple y moderno para clasificar lenguajes de especificación de SI consiste en representar en una dimensión si el lenguaje es o no es OO, y en otra dimensión si el lenguaje es deductivo (temporal) o dinámico (siguiendo la definición de lenguaje deductivo o dinámico que

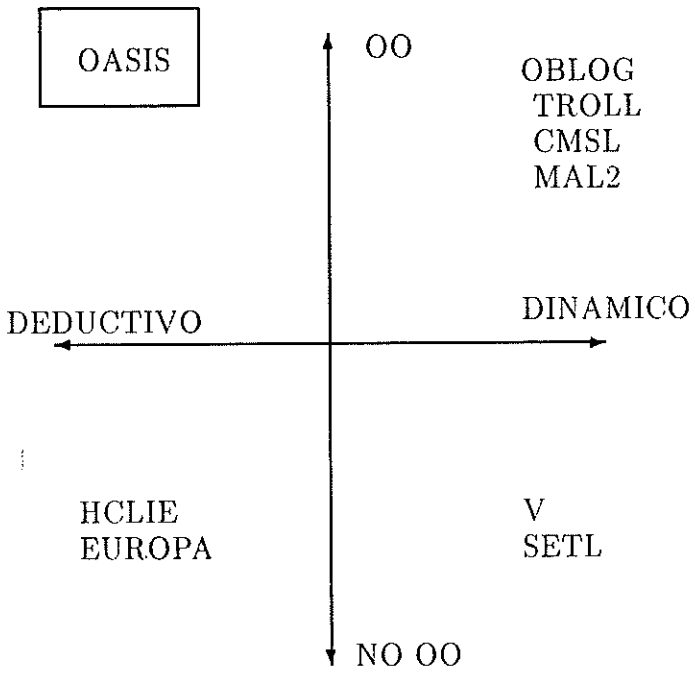


Figura 1.1: Clasificación de Lenguajes de Especificación de Sistemas de Información en dos dimensiones: Orientados o no Orientados a Objetos, y deductivos (temporales) o dinámicos.

se propone en [22] [5]), como se muestra en la fig. 1.1. Que el lenguaje sea o no OO va a estar en función de cómo se defina formalmente el modelo OO. Esto introduce de entrada un cierto grado de subjetividad en la clasificación anterior, porque es verdad que son muchas las discusiones sobre qué es realmente una aproximación OO. Pero no es menos cierto que no tiene sentido hablar de ambigüedad, porque el trabajo acumulado en los últimos años sí que permite ya disponer de un sustrato de conceptos básicos característicos de un enfoque OO. En el terreno de la Programación OO, los trabajos de Nelson [87] y Wegner [134] son una fehaciente demostración de lo dicho. Al nivel de especifi-

[5] En el capítulo 3 se recordará tal definición antes de profundizar en la vocación deductiva del Lenguaje de Especificación Oasis

cación de un Sistema de Información (SI), son diversas las propuestas que, abstrayendo los conceptos básicos asociados a la noción OO , presentan lenguajes de especificación OO para describir un Sistema de Información como una Sociedad de Objetos interactivos.

En el cuadrante OO y **deductivo** de la clasificación anterior, OASIS [90,95] es un Lenguaje de Especificación OO que, partiendo de una definición precisa del modelo OO, proporciona versiones lógico-deductivas definiendo atributos en función de eventos mediante el uso de expresividades lógicas clausales y/o ecuacionales en un entorno declarativo.

En el cuadrante OO y **dinámico** Oblog [72,117], Troll [70] y CMSL [138] entre otros adoptan un enfoque similar, proponiendo un modelo OO de la realidad, pero haciendo explícito en la especificación el efecto inmediato de un evento sobre los valores de los atributos de los objetos afectados.

EUROPA ([4]) es un Lenguaje de Programación que integra programación lógica y funcional incorporando igualdad a un lenguaje lógico con una semántica operacional de SLD-Resolución plana ('Flat-SLD Resolution'), y que utilizado como lenguaje de especificación, es una propuesta **no OO** y **deductiva**.

Otra aproximación interesante situada en el cuadrante no OO y **deductivo** la constituye el lenguaje de requerimientos lógicos HCLIE ([126]), que utiliza una lógica con claúsulas de Horn enriquecida con herencia múltiple y excepciones, dentro de un entorno basado en el uso de un nuevo paradigma de producción de software cuyas propiedades esenciales son dos:

1) Generación de una teoría de requerimientos que tenga un modelo isomorfo con el Dominio del Sistema considerado.

2) Transformación correcta de esa teoría en otra formulada en términos de un lenguaje de programación, preservando el modelo.

Por último, aunque el cuadrante no OO y dinámico está fuera de lo que en esta Tesis se defiende como entorno apropiado de producción de Software, en él se sitúan Lenguajes de Especificación funcionales como OBJ ([50]), o lenguajes con una filosofía más clásica como V ([92]) o SETL ([39]), que se diferencian de los lenguajes de programación convencionales en que enriquecen sus estructuras de control típicas (bucles,

construcciones condicionales,...) con construcciones de alto nivel como iteración sobre conjuntos o uso de cuantificadores existenciales y universales, que permitan al diseñador asumir un punto de vista declarativo si el problema lo requiere.

Cualquiera de los lenguajes de especificación anteriores, y en particular Oasis, que se presentará detalladamente en este trabajo, son Lenguajes de DOO, y el cómo llegar a una especificación que se corresponda con el SI a modelar hace necesario el estudio de los aspectos metodológicos que permitan llegar a una especificación correcta a partir de los resultados obtenidos en la fase de Análisis.

Una metodología para producir automáticamente especificaciones en OASIS a partir de los modelos obtenidos del AOO ha sido desarrollada en el DSIC-UPV y será presentada detalladamente en la parte final de esta Tesis. La principal aportación de tal Metodología consiste en esa propuesta de fusión de un Lenguaje de Especificación OO con semántica declarativa precisa, con un entorno de producción de Software fiel al Paradigma de Programación Automática. Ello se consigue proporcionando también a dicho Lenguaje una semántica operacional eficaz.

La notación 'Objectcharts' [9] constituye un Lenguaje de Especificación Gráfica para Sistemas OO que combina el ciclo de vida que caracteriza el comportamiento de los objetos con una especificación declarativa (de tipo dinámico) de los servicios que dan, y que proporciona una base para razonar sobre el comportamiento del Sistema a partir de la especificación de sus componentes.

La evolución natural de los 'Objectcharts' se ha traducido en la propuesta de un método de AOO presentado en [31], que dentro de la línea clásica de la tecnología CASE utiliza tres tipos de modelos:

* Un modelo semántico para dar cuenta de los aspectos estructurales del Sistema.

* Un modelo funcional para describir en un lenguaje formal las operaciones del sistema.

* Objectcharts como modelo dinámico que describa el comportamiento de los objetos presentes en los modelos anteriores.

Dicho método tiene como origen la aproximación OMT ([98]), en la que aspectos tradicionales y OO se combinan cubriendo todas las fases del desarrollo de un Sistema. En particular, en la fase de análisis se elaboran esos tres modelos: modelo de objetos (un diagrama tipo Entidad-Relación), modelo dinámico (máquinas de estados) y modelo funcional (DFD's), pero estos modelos resultantes del Análisis pueden formar un conjunto incoherente de descripciones (efecto red). No hay forma de asegurar la consistencia entre modelos.

En [31] el modelo de objetos y el funcional utilizan un lenguaje de especificación formal (HP-SL) ([99]), en lugar de los DFD's del modelo funcional en OMT. Además, se definen los eventos en el modelo dinámico como las unidades básicas de comunicación interobjetual. Los efectos globales se definen mediante una comunicación síncrona entre objetos a través de eventos. Se introduce también en el modelo dinámico los efectos de los eventos sobre los atributos locales de un objeto. La consecuencia es que se dispone de un método de AOO que permite estudiar la coherencia entre los modelos resultantes.

En el entorno del grupo IS-CORE, el trabajo realizado durante 1990 comparando las aproximaciones OO y binaria (modelos binarios) se extienden actualmente a otras aproximaciones ([129]), y están llevando a la extensión de NIAM ([128]) con mecanismos de modularización ([124]). Como resultado, se ha obtenido un modelo conceptual OO basado en un modelo binario ([125]).

Los principios metodológicos asociados a OBLOG (Lenguaje de Especificación OO y Dinámico) son objeto también de un extenso trabajo de investigación ([117]).

1.3 Presentación de la Tesis

El título de la Tesis ("Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos") responde de forma precisa a los objetivos primordiales de este trabajo, de acuerdo con los conceptos presentados en los puntos anteriores.

En el capítulo 2 el objetivo fundamental es precisar cuál es el modelo OO subyacente al entorno de producción de Software que queremos diseñar. Para ello, en primer lugar se introducen las definiciones más

recientes sobre conceptos básicos en SI, de acuerdo con los trabajos del grupo FRISCO (IFIP WG 8.1). Dada la definición de Sistema, y vistos criterios generales de clasificación de Sistemas, se precisa cuál es el tipo de SI en los que vamos a estar interesados, y se presenta una perspectiva OO basada en tales nociones. Esta perspectiva OO del enfoque FRISCO constituye el embrión del modelo OO que queremos caracterizar.

Posteriormente se revisan los trabajos de Y.Wand que conforman una aproximación ontológica para la definición formal del modelo OO. Las aportaciones ontológicas al modelo OO esbozado anteriormente conforman el modelo Ontológico Orientado a Objetos (O^3), utilizado como soporte del resto del trabajo.

A continuación se presenta una primera propuesta de formalización del modelo objetual definido. Una clase se caracteriza como una estructura algebraica compuesta por una 7-tupla, y se definen operadores entre clases para obtener nuevas clases (complejas) a partir de clases predefinidas. Se proporciona así un mecanismo constructivo para especificar SI. Cada nueva clase compleja generada haciendo uso de un operador entre clases se caracteriza precisando las componentes de su 7-tupla asociada, que dependiendo del operador tendrá un tipo determinado de ligaduras con respecto a las clases usadas como operandos.

En el capítulo 3 se introduce el Lenguaje de Especificación OO (OASIS), que se va a utilizar como herramienta descriptiva de SI de acuerdo con el modelo O^3 definido en el capítulo anterior. Se presentan las construcciones sintácticas proporcionadas por el lenguaje para dar cuenta de los componentes básicos de una especificación OO, que son:

1) dominios o clases primitivas.

2) clases elementales.

3) clases complejas.

Los operadores entre clases proporcionados por Oasis, que nos van a permitir definir clases complejas serán los siguientes:

1) Agregación

2) Herencia

3) Asociación

4) Composición Paralela

procedentes los tres primeros del area de Modelización Semántica, y el último usado extensamente en álgebras de procesos. La elección de esos cuatro operadores es una consecuencia inmediata de la encapsulación de aspectos estructurales y de comportamiento que posibilita el uso de un modelo OO.

Ya que queremos diseñar y desarrollar un entorno de producción automática de Software, hay que detallar cómo obtenemos los programas lógicos equivalentes a una Especificación en OASIS. Esto es lo que se presenta en el capítulo 4, donde se muestra cómo podemos definir Teorías Formales de Primer Orden (Clausales, Ecuacionales o Clausales con igualdad dependiendo de la versión del Lenguaje que se utilize) equivalentes a una especificación fuente dada. Como disponemos de Lenguajes de Programación Lógicos con los que representar y manipular en un Ordenador dichas teorías formales, estamos construyendo un Programa Lógico **equivalente** a una especificación en OASIS.

La evolución temporal o animación de esas Teorías de primer orden se caracteriza formalmente de dos maneras:

* mediante técnicas de **Metaprogramación** lógica

* definiendo los eventos como operadores constructores de los soportes ('sorts') correspondientes a las clases, en un **entorno algebraico**.

Con ambos enfoques se dispone de un marco formal simple y potente para dar cuenta de los aspectos dinámicos de un Sistema.

Con todas las componentes anteriores, en el capítulo 5 nos centraremos en los aspectos metodológicos que nos permitan obtener un producto Software final correcto (equivalente formalmente a la especificación del SI). Para ello, se desarrolla una Metodología (OO-METHOD) que, siendo fiel al modelo OO, proporciona un entorno de trabajo basado en los principios del Paradigma de Programación Automática. OO-METHOD se compone de:

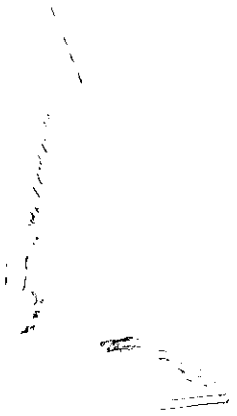
1) un método gráfico de AOO

2) traductores de alto nivel que generan de forma semi-automática la especificación en OASIS (herramienta de DOO) correspondiente a los modelos gráficos obtenidos como resultado del AOO.

3) traductores de alto nivel que, a partir de la especificación Oasis obtenida, generan automáticamente los Programas Lógicos equivalentes.

4) Animadores para dar cuenta de la evolución y comportamiento del Sistema.

Finalmente, en el último capítulo se presentan, como es habitual, conclusiones extraidas y líneas de trabajo en las que se va a seguir profundizando, así como una lista exhaustiva de las referencias bibliográficas utilizadas.



Capítulo 2

El Modelo OO

2.1 Conceptos básicos en Sistemas de Información

Son muchos los conceptos manipulados cuando se habla de Sistemas de Información, con el grave problema que sus definiciones son muy poco precisas. Se hace así difícil la comunicación técnica entre investigadores que, trabajando en un área común, utilizan en muchas ocasiones un mismo término con varias acepciones diferentes.

Ejemplos evidentes de esta situación los tenemos al contrastar las distintas definiciones de Sistema de Información, Entorno, Evento, Agente, Clase, Tipo, Sistemas Activos, ..., etc. que nos encontramos en las distintas propuestas existentes de Diseño de SI.

Con el objetivo último de sustentar el trabajo desarrollado en esta Tesis sobre un conjunto básico de nociones y definiciones aceptadas por un conjunto lo más amplio posible de la comunidad informática actual, vamos a utilizar como marco de referencia el trabajo realizado por el grupo FRISCO (IFIP WG 8.1) [76] que constituye un serio intento de proporcionar definiciones precisas para conceptos básicos en el campo de los SI.

Una vez definidos estos conceptos, daremos una interpretación OO de los mismos que nos servirá como presentación del modelo objetual que queremos caracterizar y formalizar.

2.1.1 El enfoque FRISCO

Si queremos diseñar y desarrollar SI tenemos que partir de una base filosófica común que caracterize de forma precisa la visión que tenemos del mundo real. Es cierto que se puede analizar la realidad de muchas formas diferentes, pero no es menos cierto que si se quiere hacer posible profesionalmente la comunicación de esa percepción hay que elegir un marco de referencia único.

El trabajo de FRISCO asume que formamos parte de un Universo de 'cosas' en el sentido más general. Estas 'cosas' representan cualquier concepto concreto o abstracto del mundo. Cualquier aspecto de la realidad que podemos ver, sentir, tocar o pensar de alguna forma, o cualquier abstracción o grupo de abstracciones de tales 'cosas' in-

cluido el propio mundo es considerado una 'cosa'. Aunque tenemos la impresión que se trata de 'nuestro' mundo, lo cierto es que existe independientemente de nuestra existencia. Este mundo es perceptible, podemos elaborar conceptos en nuestras mentes y pensar sobre él.

La palabra 'objeto' se reserva para denotar cosas concretas, observables con nuestros sentidos. Para referirnos a cosas abstractas utilizaremos la palabra 'concepto'.

Para poder observar partes del mundo real se introduce el concepto de **área**. Un área es una parte del mundo para la que se puede decidir de forma única si un objeto o concepto dado pertenece a dicha área o al resto del mundo. Un área puede pues ser un ente físico (p.e. un edificio) o abstracto (p.e. el conjunto de reglas por las que se rige una empresa).

La noción de área es fundamental para delimitar qué es realmente relevante cuando abordamos un análisis de una parte específica del mundo. A las cosas pertenecientes a un área les llamaremos **elementos**. P.e. en el área de una organización, empleados, departamentos y grupos de empleados de diferentes departamentos son elementos del área. Es importante destacar que la noción de elemento no se corresponde necesariamente con la de miembro de un conjunto. Como se aprecia claramente en el anterior ejemplo, dos elementos pueden ser distintos, o no disjuntos, o estar incluido el uno en el otro.

Las cosas que constituyen el mundo tienen **propiedades** que las caracterizan. El hecho de que también las propiedades sean cosas introduce una especie de recursión infinita. Pero lo realmente interesante es que la descripción que se hace de la realidad parte de un conjunto básico de nociones precisas, sobre el que se elabora de forma consistente la descripción global.

Hecha la distinción entre objetos y conceptos, es útil también introducir otro criterio de clasificación de las cosas relativo al tiempo. Primero es necesario definir el **estado** de una cosa como el estado del conjunto de propiedades que la caracterizan. Para poder hablar del estado de una cosa, ésta debe existir. La noción de existencia es muy importante. Cuando un objeto es creado aparece físicamente en el mundo. Podemos decir que existe de forma objetiva. Por el contrario, cuando se crea una concepción ésta aparece mentalmente en un sujeto (la persona). Su existencia es por tanto subjetiva.

Algunas cosas existen siempre, mientras que otras se crean en un instante determinado y son destruidas posteriomente.

Dicho todo esto, podemos hablar ya de cosas **dinámicas** y **estáticas**. Dinámicas son aquellas cuyo estado es susceptible de cambio en el tiempo, y estáticas aquellas para las que ningún cambio de estado es apreciable dentro de un periodo relevante de tiempo. Asimismo, distinguiremos entre cosas **activas** y **pasivas**. Las primeras son aquellas cosas dinámicas que son necesarias para que un cambio de estado se produzca. Las segundas son las no activas durante un periodo temporal relevante.

La anterior distinción entre dinámico-estático y activo-pasivo depende de dos aspectos

* el conjunto de propiedades que caracteriza el estado de la cosa

* el intervalo de tiempo en el que dichas propiedades son observadas

Por tanto, una cosa dada puede ser vista como estática/pasiva en un cierto intervalo, y como dinámica/activa si el periodo elegido es mayor o diferente. P.e., las normas de funcionamiento de una organización puede ser algo estático si se considera un intervalo de tiempo en el que no se han producido cambios, o dinámica si el intervalo es mayor e incluye alguna modificación de dichas normas.

En este contexto, un **proceso** es una cosa activa que es observada como la inductora de la creación de otra cosa, o de un cierto cambio en su estado. Puede ser discreto en cuyo caso se le llama **actividad**. Está compuesto de partes más elementales llamadas **operaciones**. Una **operación** es una parte elemental de un proceso causante de la creación de una cosa o de la modificación elemental de una o más de sus propiedades.

El concepto de operación es relativo y depende del nivel de abstracción utilizado. Lo que a un nivel puede verse como un proceso, a un nivel de abstracción mayor puede ser una operación.

Se define el **comportamiento** como la evolución dinámica de una cosa. La noción de comportamiento sólo tiene sentido si existe algo con identidad propia que evoluciona a través del tiempo cambiando su estado dependiendo de la ocurrencia de procesos.

2.1.2 ¿Qué es un Sistema?

Una vez presentado el entorno básico en el que nos movemos, vamos a abordar la definición de Sistema a partir de una interpretación no tradicional, en la que un Sistema no va a ser visto como algo absoluto, sino como una concepción de la realidad realizada por lo que denotaremos como **observador del Sistema**. Esto induce una distinción entre Sistema y Dominio del Sistema, e introduce la noción que las propiedades características del Sistema son **subjetivas** y están asociadas al Dominio del Sistema por el Observador cuando realiza su observación.

El término Sistema de Información (SI) parece ser usado habitualmente con cuatro acepciones distintas:

* Referido a un moderno Sistema de Proceso de Datos creado en torno a un Sistema de Gestión de Bases de Datos (SGBD) utilizando la tecnología informática existente en el mercado

* Como una abstracción de un Sistema como el anterior en la que se omite todo aspecto relacionado con la implementación. En terminología FRISCO se trata de un Sistema de Información Restringido (SIR).

* Referido al conjunto de actividades realizadas en una organización con el fin de intercambiar y dar soporte a la informacion de dicha organización. FRISCO se refiere a estos Sistemas como Sistemas de Comunicación (SC).

* Como una concepción global de toda la manipulación de datos y comunicación de actividades de una organización. Esta acepción incluye a las tres anteriores como sub-sistemas.

Un Sistema va a ser definido como una concepción de un área (**Dominio del Sistema**) cuyos elementos aparecen relacionados de forma que constituyen un 'todo' homogeneo con al menos una propiedad característica con respecto al entorno. Esta propiedad no la posee ninguno de los elementos.

El **Observador del Sistema** es la persona que concibe el Dominio del Sistema como un Sistema.

El **entorno** de un Sistema está formado por todas aquellas cosas del mundo que no forman parte de dicho Sistema, pero que son necesarias si queremos describir las propiedades características del mismo.

El **Dominio del entorno** es el área que se concibe como entorno del Sistema.

El término **Vista del Sistema** se refiere al conjunto de elementos del Dominio del Sistema, del Sistema, del Entorno y del Dominio del Entorno, y a las relaciones entre estos elementos, que son relevantes para que el Observador del Sistema pueda describir el Sistema considerado.

En este contexto, diseñar e implementar un Sistema va a consistir en agrupar y estructurar cosas de una parte del mundo real (que se convierte así en el Dominio del Sistema) con el objetivo de dotarles de ciertas propiedades características del Sistema.

Definición 2.1 *La noción de Sistema propuesta por FRISCO se puede formalizar como:*

$$S_f = (A, VS)$$

donde A es un área arbitraria y VS es la vista del Sistema expresada como:

$$VS = (S, E, DS, DE, PCS)$$

donde:

* *S es un Sistema*
* *E es el entorno del Sistema*
* *DS es el Dominio del Sistema (el área concebida como Sistema)*
* *DE es el Dominio del Entorno (el área concebida como Entorno)*
* *PCS son las Propiedades Características del Sistema, que constituyen una relación entre el Sistema y el Entorno.* □

2.1.3 Clases de Sistemas

Aunque resulte extraño, son pocos los intentos que se han realizado para elaborar una taxonomía de Sistemas [1,16,27]. A continuación se presentan unos criterios de clasificación que nos llevarán a la categoría de Sistemas Abiertos y Activos, que serán los estudiados a lo largo del trabajo.

En la Teoría General de Sistemas, éstos pueden ser clasificados como:

* estáticos o dinámicos

* cerrados o abiertos

* pasivos o activos

Un Sistema **dinámico** es aquél en el que el comportamiento de alguna parte del Dominio del Sistema o del Dominio del entorno forma parte de la Vista del Sistema. Si ésto no ocurre, el Sistema es **estático**.

Un Sistema **activo** es un Sistema dinámico en el que un proceso en el Dominio del Sistema es parte del Sistema. Un Sistema es **pasivo** si ésto no ocurre. Un Sistema estático es necesariamente un Sistema pasivo.

Por último, un Sistema **abierto** es un Sistema Dinámico en el que un cambio de estado en el Dominio del Sistema puede ser causado por el entorno, o en el que un cambio de estado en el Dominio del Entorno se concibe como causado por un proceso en un Sistema Activo. Cuando no se cumple lo anterior el Sistema será **cerrado**. Un Sistema estático es forzosamente un Sistema cerrado.

Las restricciones existentes entre estos distintos criterios de clasificación reducen el número de Sistemas que van a ser considerados a los cuatro siguientes:

1) Sistemas Estáticos

2) Sistemas Activos Cerrados. Temporalmente son dinámicos pero acabarán siendo estáticos debido, por ejemplo, a la 2 Ley de la Termodinánica.

3) Sistemas Abiertos y Pasivos (SAP), que son dinámicos debido exclusivamente al comportamineto del entorno.

4) Sistemas Abiertos y Activos (SAA), dinámicos al menos como consecuencia de su propio comportamiento.

A lo largo del trabajo se van a presentar entornos de producción de software para diseño de SI Abiertos y Pasivos (MOL), y sus extensiones activas (OASIS). En el contexto SAA, el comportamiento de un Sistema se va a manifestar a través de:

* **funciones internas**: los cambios en el Dominio del Sistema son producidos por procesos del mismo Dominio.

* **funciones externas**: hay que distinguir dos situaciones:

  1) **impresiones**:los cambios en el Dominio del Sistema se conciben como originados por el Entorno

  2) **expresiones**: se producen cambios en el Dominio del Entorno provocados por el Sistema

Los SAA pueden clasificarse de varias formas según su comportamiento. Dentro del marco de FRISCO distinguiremos cuatro tipos en función de las siguientes consideraciones:

* Una **reacción** de un SAA es una expresión causada incondicionalmente por una impresión.

* Una **acción** de un SAA es una expresión vista como absolutamente independiente de cualquier impresión.

Podemos definir ahora los cuatro tipos de SAA:

1) **Sistemas Encapsulados**: son SAA en los que no existen expresiones (cambios de estado en el Dominio del Entorno). El entorno actúa a la vez como observador e inductor de procesos, generando impresiones y funciones internas.

2) **Sistemas Reactivos**: son SAA en los que cada expresión es una reacción, y en los que una impresión origina inmediatamente una reacción.

3) **Sistemas Sensibles**: son SAA (también posiblemente reactivos) en los que se satisface que para al menos una expresión, una cierta impresión o conjunto de impresiones es una condición necesaria pero no suficiente para que la expresión ocurra.

   Un ejemplo clásico lo constituye la recepción de un pedido en un Sistema de Ventas como impresión necesaria pero no suficiente para producir la expresión 'distribúyase lo pedido'.

4) **Sistemas Autónomos**: son SAA (posiblemente sensibles, pero no reactivos) en los que al menos una expresión es una acción.

   Un ser humano y la práctica totalidad de las organizaciones pueden ser consideradas Sistemas Autónomos.

2.1.4 Sistemas Organizacionales

El conjunto de Sistemas agrupados en la categoría de los SAA es demasiado amplio. Un mosquito, una cebolla, un volcán, una constelación, un ser humano son ejemplos muy claros de lo diverso que es el conjunto de los SAA.

En este trabajo nos vamos a interesar por un subconjunto muy particular de SAA que son los llamados **Sistemas Organizacionales**. Definamos en primer lugar la noción de Dominio Organizacional.

Un **Dominio Organizacional** (DO) es un área de la Sociedad Humana incluida en lo que denotamos habitualmente como organización, fábrica, compañía, empresa, institución (pública o privada), comercio etc., o cualquiera de sus partes como plantas de producción, departamentos de ventas, secciones de personal, contabilidades etc.

Cada uno de estos DO's pueden verse como SAA que interaccionan con un entorno que corresponde a una parte de la Sociedad. A este tipo de Sistemas es a lo que vamos a llamar Sistema Organizacional.

Un **Sistema Organizacional** (SO) es un SAA concebido a partir de un DO. Un SO se manifiesta a través de expresiones cuya finalidad es satisfacer o cambiar el comportamiento del entorno, bien como respuestas a impresiones, bien a través de un comportamiento autónomo.

La abstracción y especialización que nos lleva de la noción general de Sistema vista anteriormente a la de SO se conoce con el nombre de

abstracción organizacional (AO). Esta AO introduce nuevos conceptos en el marco FRISCO, como son los de actor, actividad, evento[1], disparo y comunicación.

Podemos expresar informalmente esta AO definiendo un SO en el contexto FRISCO como:

Definición 2.2 $SO_f = (SAA_f, AO)$

donde SAA_f referencia el conjunto de conceptos asociados a los SAA, y AO es la Abstracción Organizacional, que expresa una restricción y especialización de las componentes S y DS de la definición de Sistema dada anteriormente, desde el nivel de los SAA. En concreto:

$$AO = (C, R, PCSO)$$

donde:

* $C = (A, P, O, E, T)$ *con*

  1) *A:Actores (o Agentes). Son elementos del DO concebidos como responsables de la ejecución de actividades en el SO*
  2) *P:Actividades. Una actividad es un proceso* **discreto** *en el DS que es la causa de un cambio de estado en dicho dominio*
  3) *O:Operandos de las actividades. Son objetos o conceptos necesarios para la correcta ejecución de la actividad.*
  4) *E:Eventos, que permiten expresar las dependencias dinámicas entre actividades. Se define un evento como una abstracción de un cambio de estado. Es discreto, no tiene duración y ocurre en un cierto instante. Cualquier evento puede actuar como* **disparador**, *siendo un disparador una relación entre un evento y una o más actividades que a un cierto nivel de abstracción indica a los agentes adecuados que tales actividades deben ejecutarse.*
  5) *T:Instantes en el tiempo*

[1]Usarenos el término evento para designar al concepto de 'event' en lengua inglesa. Aunque evento no es formalmente un término castellano, su amplia divulgación lo convierte en la práctica en un anglicismo aceptado

* *$R = (Ag, Dp, In, Ou, Oc)$ define las siguientes cinco relaciones básicas:*
  - *Ag representa el papel de agente. (A, P) significa que un actor A realiza la actividad P.*
  - *Dp son relaciones de disparo. (E, P) indica que un evento E dispara la actividad P.*
  - *In denota relaciones de entrada para actividades. (O, P) indica que O es operando de entrada de la actividad P.*
  - *Ou referencia relaciones de salida. En este caso (O, P) indica que es O es operando de salida de la actividad P.*
  - *Oc son relaciones de ocurrencia. (E, t) informa que el evento E ha ocurrido en el instante T.*
* *PCSO define las Propiedades Características del SO considerado.*□

En resumen, la AO que nos lleva a los SO tiene como características más relevantes las siguientes:

* Deja de lado todos los aspectos referentes a transformación de energía (p.e. producción o consumo de calor, metabolismos de organismos vivos,etc.).
* Sólo se consideran procesos discretos y finitos (actividades).
* Toda actividad debe ser realizada por un actor o agente explícito.
* Toda actividad tiene un objetivo, cuyo resultado final es una contribución a algunas de las propiedades características del SO.
* Los agentes deben intercambiar conocimiento sobre la existencia y estado de los componentes del SO. Un agente puede ejecutar la actividad de enviar un conjunto de conocimiento a otro que ejecutará la actividad de recepción.
* Las actividades deben de ejecutarse con un cierto orden dinámico. Para sincronizar actividades relacionadas dinámicamente entre sí se introducen los conceptos de evento y disparador.
* Los operandos de las actividades pueden ser concretos (objetos) o abstractos (conceptos que aparecen como conocimiento).

2.1.5 Conceptos básicos de clasificación

Resulta obvia la necesidad de agrupar y clasificar las cosas que componen un Sistema, y de describir estos grupos y diferenciarlos entre sí.

Los clásicos conceptos matemáticos de **conjunto, subconjunto, elemento** y **relación** son adoptados por FRISCO como conceptos básicos.

La definición precisa del concepto de **tipo** y de **clase** requiere una mayor atención. Algunos Lenguajes de Programación como SIMULA utilizan el concepto de clase de una forma específica, mientras que la noción de tipo es utilizada frecuentemente para declarar qué valores o estructuras de datos pueden ser asignados a variables en programas.

En el Lenguaje Natural, el término clase suele denotar un conjunto cuyos elementos comparten ciertas propiedades, pero frecuentemente el término tipo se utiliza en el mismo sentido.

FRISCO propone las siguientes definiciones para estos conceptos:[2]

* Un **tipo** se define como un conjunto de propiedades que caracterizan de forma conjunta ciertas cosas presentes en un área determinada

* Un **subtipo** S de un tipo T es un conjunto de propiedades tales que T es un subconjunto de S.

* Un **supertipo** es un subconjunto de un tipo.

* Una **clase** es el conjunto de cosas de un cierto tipo presentes en un área.

Es importante resaltar una vez más que el concepto de tipo es relativo a un área. Un tipo relevante en un área dada puede no serlo en otra. Lo mismo ocurre con la noción de clase como aglutinador de elementos de un tipo dentro del área considerada.

Ciertos operadores lingüísticos pueden ser definidos a este nivel. Por ejemplo, el operador **instancia de**, que opera sobre un tipo y devuelve

[2]Posteriormente veremos de qué forma se definen estos conceptos en el modelo OO que queremos caracterizar.

un miembro de su clase. Otro operador conceptual es **población de**, que dado un tipo nos devuelve su clase. En este contexto, se verifican las siguientes propiedades:

* Una clase es la *población* de su tipo correspondiente
* Un miembro de una clase es una instancia del tipo asociado a la clase.
* La población de un subtipo S de un tipo T es un subconjunto de la población de T.
* Un tipo constituye una descripción intensional de la clase correspondiente.

Meta-tipos pueden ser definidos a partir de un tipo dado. Su uso es útil porque a menudo es necesario discutir aspectos de clasificación de ciertos conceptos en dos diferentes niveles de abstracción. P.e., al nivel conceptual en el que estamos definiendo conceptos básicos en SI, tiene pleno sentido hablar de los tipos *entidad*, *actividad* etc. Si nos posicionamos en el nivel de abstracción propio de un Analista de Sistemas, *Libro*, *Socio* etc. son tipos del metatipo Entidad.

No hay que confundir la relación entre metatipo y tipo con la relación existente entre tipo y subtipo. En el primer caso se está aplicando un principio de abstracción, mientras que en el segundo caso se aplica el principio de generalización (organización de cosas según ciertas propiedades en el mismo nivel de abstracción)

Una vez analizada la esencia del enfoque FRISCO en la normalización de los conceptos básicos en SI, y fijado nuestro contexto de trabajo en los SO, vamos a presentar una propuesta de definición de Conceptos de SI desde una perspectiva Orientada a Objeto.

2.1.6 Una perspectiva OO del enfoque FRISCO

Hemos definido en el punto anterior un Sistema en su acepción más general como una colección de elementos (el **Dominio del Sistema**) que posee al menos una propiedad característica con respecto al entorno que no tiene ninguno de los elementos por separado. Definíamos también una **Vista del Sistema** como el conjunto de elementos del

Dominio del Sistema, el Dominio del Entorno y de relaciones entre estos elementos necesarias para describir el Sistema.

En el entorno de los SO (especialización de SAA, como acabamos de ver), la tarea de un Analista es especificar una vista del Sistema para obtener una especificación formal del Sistema Objeto. Vamos a considerar que un Sistema real está compuesto por una sociedad de objetos que interaccionan entre sí. Las 'cosas' de las que hablábamos en el sentido más general de la palabra en el punto anterior van a ser algo mucho más concreto: **objetos**.

El uso de una perspectiva OO es muy útil porque los conceptos que vamos a manipular desde el punto de vista de la especificación del Sistema, van a estar muy próximos a los conceptos básicos de SI que hemos definido. De esta forma el conocido 'gap' semántico o diferencia entre la percepción que tenemos del SI objeto y la representación que de él se hace, se reduce sustancialmente.

Un objeto es una unidad operacional auto-contenida, que va a encapsular aspectos estáticos y dinámicos. Las propiedades de los objetos se representan por medio de **atributos**. No todas las propiedades de un objeto van a ser relevantes. El conjunto de propiedades seleccionadas (atributos del objeto) va a depender del Analista (Observador del Sistema), que elegirá aquéllas que estime necesarias de acuerdo con el propósito de su modelización.

Un objeto tiene asimismo un estado que viene dado por el conjunto de los valores de sus atributos en un instante dado. Cuando un objeto se crea inicia su vida, que finalizará cuando sea explícitamente destruido. Durante su existencia, todo objeto tiene un nombre que lo identifica unívocamente.[3] Asimismo, un objeto tiene propiedades estáticas y dinámicas según no cambien o sí lo hagan respectivamente durante su vida. P.e., entre los atributos de un objeto podremos distinguir entre constantes y variables en función de que se correspondan

[3]Es importante destacar que estamos considerando la identificación de un objeto como una propiedad más del objeto. La polémica sobre si la distinguibilidad de un objeto reside en un 'algo'especial que engloba a todas las demás propiedades versus la idea de que un objeto es un conjunto de propiedades [79] es así solventada adoptando esta última aproximación. En nuestro modelo OO de la realidad no existen objetos indistinguibles, porque al menos siempre una propiedad (su identificador) los diferencia. Otros trabajos [138] adoptan la otra postura.

con propiedades estáticas o dinámicas.

La dinámica de nuestra Sociedad de Objetos está caracterizada por el concepto de **evento**. Un evento se define como la abstracción de un cambio de estado, de modo que los objetos cambian de estado debido a la ocurrencia de eventos. Cada objeto tiene asociado un conjunto de eventos, que son los que le pueden afectar durante su vida. El estado de un objeto en un instante dado será una función de los eventos que conforman su vida hasta ese momento en el que es observado. Un objeto se convierte de esta forma en un proceso observable.

Los eventos son activados por **agentes**, que son los responsables de su ejecución. Se introduce actividad en la Sociedad de Objetos permitiendo que un objeto actúe como agente (generador de eventos) cuando determinadas condiciones en él definidas se satisfagan. Estas condiciones se llaman **relaciones de disparo**.

Los objetos del Sistema no son independientes. La interacción interobjetual se produce de dos formas:

* por compartición de eventos. Un evento compartido es aquel que afecta a la vida de más de un objeto.

* mediante las relaciones de disparo

Por medio del clásico mecanismo de abstracción de clasificación, agrupamos objetos con las mismas propiedades en **tipos**. Se define una **clase** como el conjunto de propiedades que caracterizan a un grupo de objetos. Dicho de otra forma, una clase es una caracterización precisa de un conjunto de propiedades estructurales y de comportamiento que una colección de objetos comparte. Esta colección de objetos constituye el tipo[4].

Podemos definir dos operadores: un operador 'instancia de' que tiene como operando una instancia de objeto (un elemento del tipo) y devuelve la clase a la que pertenece. Y un operador 'universo de', que dada una clase devuelve su tipo correspondiente.

[4]Es necesario reseñar que el concepto de tipo y clase no es el mismo en muchas propuestas de modelos OO. Sin ir más lejos, FRISCO propone la definición opuesta como se ha visto. El hecho de denotar la población de una clase con la noción de tipo se sustenta en el concepto histórico de tipo de datos, en el que al definir el tipo lo que se está dando es el conjunto de valores que lo conforman.

Toda Sociedad de Objetos se construye a partir de un conjunto de **clases primitivas**, constituidas por objetos que tienen un único estado y que siempre existen (entidades 'platónicas').

A las clases no definidas a partir de otras las llamaremos **clases elementales.**Una vez definidas estas clases elementales, debe darse un mecanismo constructivo que nos permita dar cuenta de clases complejas (definidas a partir de otras clases). Para ello introducimos operadores de clases, procedentes de los mecanismos de abstracción utilizados en Modelización Semántica (agregación, especialización, asociación) más un operador de composición paralela que permita componer objetos vistos como procesos. De esta forma, en última instancia, la Sociedad de Objetos considerada como un todo, es el tipo de una clase compleja construida por composición paralela del resto de las clases definidas.

Llamaremos **esquema conceptual** a la descripción de las componentes de un SO, incluyendo sus clases (primitivas, elementales y complejas) y las propiedades de cada una de esas clases (atributos, eventos, relaciones de disparo, etc.).

Esta visión OO de un SI, enriquecida con aportaciones del modelo objetual formal propuesto por Wand [130] que vamos a analizar a continuación, constituirá el esqueleto del entorno de producción de Software presentado posteriormente.

Como comentarios finales, es interesante resaltar que a pesar de que el modelo OO aparece históricamente ligado a Lenguajes de Programación OO (LPOO), la aportación de esta propuesta es que introduce conceptos a un nivel de especificación, que permiten describir SO de una forma muy simple y rica a la vez, y que mantiene una gran similitud con los conceptos clásicos en LPOO. En particular, la noción de atributo es la misma, y los mensajes tienen una interpretación muy simple, pero distinta dependiendo si son mensajes de actualización o de consulta.

Los mensajes de actualización se corresponden con los eventos, siendo el método asociado a un mensaje el correspondiente cambio de estado inducido por el evento en el objeto(s).

Como un objeto es un proceso observable en cualquier instante, los mensajes de consulta se corresponden en el modelo con la ejecución de observaciones sobre el objeto cuyo estado se quiere conocer.

2.2 Un modelo OO basado en conceptos ontológicos

2.2.1 Equivalencias entre conceptos ontológicos y conceptos tradicionales OO

Wand presenta en [130,131,132] una propuesta formal de modelo objetual basada en la Ontología [20]. La Ontología es la rama de la Filosofía de la Ciencia que estudia la modelización de la existencia de las cosas en el mundo. El trabajo de Wand parte de la idea de utilizar una perspectiva ontológica para sentar una base formal de la noción de objeto. Si los objetos son la estructura básica de modelización de un cierto dominio del Sistema, ¿por qué no acudir a la rama de la Filosofía que aborda este mismo problema?. En esta linea de razonamiento, la posición defendida en esta Tesis es clara: la cada vez mayor relevancia del modelo objetual es una indicación empírica de que para cualquier persona es más facil describir sus percepciones del mundo a través de la noción de objeto.

La relación entre conceptos ontológicos y conceptos típicamente OO se establece fácilmente. Vamos a considerar como principios básicos del modelo objetual los cinco propuestos por [88]: abstracción de datos o encapsulación, independencia y persistencia, comunicación entre objetos (intercambio de mensajes), herencia y homogeneidad.

* El concepto de abstracción de datos es fundamental en un modelo OO para centrarse en el comportamiento del objeto despreocupándose de aspectos relacionados con su representación. En los modelos clásicos, los objetos encapsulan un conjunto de variables de instancia (atributos) y un conjunto de operaciones (métodos) que actúan sobre ellas.

  En la aproximación ontológica los objetos cambian y estos cambios no pueden ser descritos independientemente de los objetos. Desde este punto de vista, el concepto de encapsulación es fundamental. Sin embargo, desaparece el concepto de método o de cualquier otro mecanismo que cambie el estado de los objetos. En su lugar, la noción de *ley* define el conjunto de estados permitidos

o prohibidos, siendo estas leyes propiedades de los objetos.

* La noción de independencia hace referencia a dos características de un objeto: la primera relacionada con el estado del objeto, y la segunda con su existencia [130]. Como se explica en [88], los objetos controlan su propio estado. Una vez creados, un objeto continua existiendo (o persistiendo) aunque su creador deje de existir.

  La acepción clásica de independencia establece que un objeto sólo puede cambiar su estado a través de las acciones propias de sus métodos, es decir, por medio de sus características dinámicas. Por tanto, la encapsulación es una condición necesaria para poder hablar de independencia.

  Desde el punto de vista ontológico, los posibles estados de un objeto vienen descritos por sus leyes, que son propiedades del objeto, con lo que se da plena cobertura a dicho concepto de independencia. La persistencia se manifiesta a través del *principio de invariancia nominal*, que posibilita que cambie el estado de un objeto sin que cambie su 'esencia'.

* En las aproximaciones tradicionales, los objetos sólo se comunican mediante el intercambio de mensajes. Los métodos se ven así como respuestas a posibles mensajes.

  La Ontología establece como principio que los objetos interaccionan entre sí. El efecto de esta interacción se manifiesta a través de la historia (vida) de los objetos, en particular a través de los estados por los que el objeto va pasando. Desde el punto de vista ontológico, no se habla de en base a qué mecanismo(s) esta interacción se produce.

* La noción clásica de homogeneidad implica que todo en el modelo es un objeto. Aparece así una circularidad que debe ser rota de forma arbitraria en algún nivel.

  Por el contrario, la Ontología efecúa una clara distinción entre los objetos y sus propiedades. No existe, pues, una completa homogeneidad. Además las propiedades no pueden existir sin los

objetos. Los objetos pueden estar compuestos de otros objetos, pero se asume que existen objetos elementales no descomponibles. De esta forma la existencia de un nivel básico es un principio, no una decisión arbitraria. Algunas propiedades pueden corresponder a un objeto que es parte componente de otro.

Asimismo, se introduce la distinción entre propiedad heredada y emergente, formalizando la idea de que un objeto compuesto puede ser totalmente diferente al conjunto de objetos que lo componen.

* Los objetos se agrupan en tipos/clases. Los objetos pueden especializarse en subtipos/subclases, de forma que el conjunto de objetos de una subclase está incluido en en el conjunto de objetos de la clase padre. Este mecanismo de especialización conforma una jerarquía IS-A, de forma que de acuerdo con el enfoque tradicional, los objetos de la subclase heredan variables de instancia y métodos de la clase padre.

  Bajo la aproximación ontológica, las clases y tipos se definen usando la noción de alcance de una propiedad (conjunto de objetos que la poseen). El alcance de un conjunto determinado de propiedades (las que definen una clase) nos devuelve el tipo de la misma. Por ello, tanto las propiedades estáticas (atributos) como las dinámicas (leyes) son utilizadas para definir jerarquías de clase.

Vista esta correspondencia entre conceptos objetuales y ontológicos, vamos a analizar brevemente el modelo objetual ontológico propuesto por Wand.

2.2.2 Un Modelo Ontológico y OO

Aspectos estáticos

El modelo propuesto por Wand considera que el mundo esta compuesto por **objetos** que describen entidades concretas en lugar de tipos o clases. P.e., al hablar de personas, cada una es un objeto individual.

El conocimiento que se tiene de los objetos del mundo proviene de sus **propiedades**. Las propiedades son intrínsecas a los objetos,

es decir, un objeto las posee con independencia de que nosotros las conozcamos o no.

Sin embargo, los **atributos** se definen como características asignadas a los objetos por un observador humano. Las propiedades se representan mediante estos atributos. Por ejemplo, asignamos un atributo *color* a un objeto como representación de su propiedad física de reflejar ciertas longitudes de onda de una radiciación electromagnética.

Los objetos no pueden ser separados de sus propiedades. Estas propiedades pueden ser de dos tipos: **atributos observables** y **leyes**. Los primeros son funciones que, dado un objeto que posee una propiedad determinada, me devuelven un valor de un cierto dominio asociado a la propiedad. Es decir, son propiedades a las que se les puede asignar valores específicos.

Los valores de los atributos de un objeto pueden estar restringidos por una serie de reglas que llamaremos leyes. Son por tanto restricciones sobre las combinaciones posibles de los valores de los atributos de los objetos. Estas leyes reflejan restricciones naturales o artificiales asociadas a los objetos.

Como ambos son propiedades, se concluye que la representación de una propiedad puede ser bien una función que asigna un valor a la propiedad considerada, bien un predicado que especifique si las restricciones de estado se satisfacen. Existe no obstante una diferencia importante entre ambas: las restricciones afectan a todo un tipo (están cuantificadas universalmente), mientras que las propiedades hacen referencia a un solo elemento de un tipo (cuantificación existencial).

El conjunto de propiedades utilizado para describir un cierto objeto es subjetivo: depende del punto de vista y del propósito de la modelización. [5] Este conjunto constituye el **Esquema Funcional** del objeto.

El valor de un atributo en un instante dado es el de una **variable de estado**, y el conjunto de valores de las variables de estado define el **estado** del objeto.

El conjunto de estados permisibles para un objeto viene caracteri-

[5] Esta subjetividad en el Análisis del mundo sintoniza completamente con el modelo de FRISCO, donde el Observador del Sistema elige de forma también subjetiva qué propiedades del Sistema Objeto van a ser relevantes.

zado por el conjunto de sus leyes.

Un objeto que puede ser descrito como asociación de otros es un **objeto compuesto**. En caso contrario se trata de un **objeto elemental**. Las propiedades de un objeto compuesto son de dos tipos:

1) **resultante** o **hereditaria**, cuando la propiedad considerada pertenece a alguno de los objetos componentes.

2) **emergente**, cuando la propiedad en cuestión es una nueva propiedad del objeto como objeto compuesto, y no pertenece a ningún objeto componente.

Aspectos Dinámicos

Los objetos sólo pueden permanecer en estados válidos. Cuando un objeto queda en un estado inválido, debe cambiar de forma espontánea hacia un estado válido, definido de forma única por el estado inválido en cuestión (de acuerdo con las restricciones).

Un objeto no puede alcanzar un estado inválido de forma espontánea. P.e, si consideramos una cuenta bancaria y sus posibles transacciones, cuando una de estas transacciones ocurre el saldo de la cuenta deja de ser la suma de todas las transacciones y el objeto *cuenta* queda en un estado inválido. La espontánea evolución al estado válido correspondiente modificará el nuevo valor del saldo de la cuenta. Este saldo no puede cambiar de forma espontánea como se ha dicho antes, sino tan sólo cuando una transacción ocurre.

Es importante resaltar que los objetos no obedecen leyes, sino que lo que ocurre es que las leyes fijan los estados que los objetos pueden asumir. Una situación en la que un objeto permanece en un estado no válido indica que la ley no está bien definida. Bajo este punto de vista, los objetos son entes reales. Si los objetos son vistos como implementaciones la situación cambia, ya que se dan las leyes y el objeto es implementado de forma que las debe obedecer.

La interacción entre objetos en este modelo se da por medio de leyes que relacionan variables de estado de objetos distintos. De esta forma un objeto puede afectar al estado de otro objeto.

Si un objeto puede afectar el estado de otro, puede dejarlo en un estado inválido. En este caso, el objeto afectado deberá cambiar su

estado y evolucionar a un estado válido.

Tenemos pues dos tipos de leyes: intraobjetuales e interobjetuales. Formalmente,

Definición 2.3 *Si $S(i)$ denota el conjunto de estados de un objeto i, una ley intraobjetual es una función sobre el conjunto de estados:*

$$L : S(i) \rightarrow S(i)$$

* *Un estado tal que $L(s) = s$ es un* estado válido

* *Un estado tal que $L(s) \neq s$ es un* estado inválido

Definición 2.4 *Si i,j son dos objetos, una ley interobjetual es una función:*

$$A : S(i) \times S(j) \rightarrow S(i) \times S(j)$$

* *Sea s^k el estado de un objeto k. Un estado tal que $A(s^i, s^j) = (s^i, s^j)$ es un* estado interobjetual estable.

* *Un estado tal que $A(s^i, s^j) \neq (s^i, s^j)$ es un* estado interobjetual inestable

Si dos objetos no interaccionan, entonces cada uno de ellos puede encontrarse en cualquiera de sus estados válidos. Pero si interaccionan, entonces no serán estados correctos todas las posibles combinaciones de sus respectivos estados válidos, debido a la existencia de estas leyes interobjetuales.

Estas leyes contienen la información sobre potenciales cambios de estado, y cuales serán dichos cambios de estado. Es decir, la dinámica de la Sociedad de Objetos queda caracterizada por las leyes, que determinan los cambios de estado de los objetos.

Una transición entre dos estados recibe el nombre de **evento**. Formalmente, vamos a definir un evento como un par ordenado de estados:

Definición 2.5 *Dado un objeto i, un evento e es un par ordenado de estados $e = (s, s')$ con $s, s' \in S(i)$*

No todo par de estados que representa un evento ocurre necesariamente, pero sí que es cierto lo contrario: todo evento puede ser descrito como un par de estados.

Como una ley establece los cambios de estado posibles, puede ser dada de forma explícita enumerando los eventos que comienzan en estados inválidos, ya que los estados válidos no son modificados por la ley.

Consideremos como ejemplo un caso en el que dos objetos i,j se encuentran en un estado inestable (s_1^i, s_1^j). Supongamos que el objeto i está en un estado estable, y que debido a la interacción con el objeto j (una ley interobjetual entre i y j), i cambia al estado s_2^i. Si s_2^i es un estado válido, no ocurre nada más. Si s_2^i es un estado inválido, entonces el objeto i cambiará a un nuevo estado s_3^i que será legal de acuerdo con sus leyes. La acción del objeto j sobre el objeto i se modeliza por tanto como la ocurrencia de dos eventos: (s_1^i, s_2^i) debido a la ley interobjetual, y (s_2^i, s_3^i) debido a las leyes intraobjetuales.

La noción de interacción entre objetos se formaliza en este modelo ontológico de la siguiente forma:

Definición 2.6 *Un objeto i afecta a otro objeto j sii el estado de i es relevante para las transiciones de estado de j.*

Diremos que i,j **interaccionan** *sii i afecta a j o j afecta a i*

Para que dos objetos interaccionen, las leyes que gobiernan sus transiciones deben de tener al menos una variable de estado en común.

2.2.3 Aportaciones del modelo OOO:el modelo O^3

Partiendo de los conceptos dados por FRISCO, se ha presentado en 2.1.6 una primera propuesta de definición de modelo OO dentro del ámbito de los SO. El enriquecimiento de este modelo con conceptos del modelo OO y ontológico de Wand que acabamos de analizar nos va a permitir fijar los conceptos básicos de nuestra propuesta de definición del modelo objetual: **el modelo O^3 (Modelo Orientado a Objeto y Ontológico)**.

La propuesta de Wand sintoniza en un porcentaje muy elevado con el modelo OO presentado en 2.1.6. Los aspectos estáticos de los objetos

vienen dados por los atributos, a los que añadimos restricciones de integridad estáticas, que son las leyes estáticas propuestas por Wand para fijar las combinaciones de valores de atributos aceptadas como válidas para cualquier estado. Asimismo restriciones de integridad dinámicas (leyes dinámicas en el formalismo de Wand) pueden introducirse para establecer relaciones entre valores de atributos en diferentes estados.

La no homogeneidad de la propuesta de Wand es heredada también por O^3 en el sentido que no todo van a ser objetos. La noción de propiedad rompe esa ambigua circularidad presente en la mayoria de propuestas OO tradicionales. Existen objetos elementales (no compuestos de ningún otro objeto) que poseen sus propias propiedades, y a partir de los cuales vamos componiendo las clases complejas presentes en el Sistema.

El formalismo de Wand proporciona una base sólida al concepto de operador entre clases, ya que los objetos complejos generados con tales operadores van a tener tanto propiedades heredadas (de sus componentes) como emergentes (nuevas propiedades del objeto compuesto como objeto componente de la Sociedad de Objetos).

Con respecto a la dinámica del Sistema, el concepto de evento como abstracción de un cambio de estado va a continuar siendo la piedra filosofal del modelo. Las leyes del modelo de Wand que dan cuenta del comportamiento dinámico de la Sociedad de Objetos van a ser simplificadas en O^3 porque sólo van a ser aceptados eventos que dejen a los objetos participantes en estados válidos.

Esa no arbitraria ejecución de eventos está controlada por la noción de **precondición** que se asocia a cada evento del Sistema. Las precondiciones van a ser leyes que restrinjan la potencial ejecución de eventos. Sólo si un evento satisface su correspondiente precondición (es decir, si conduce al objeto(s) participante(s) a un estado válido), va a ser considerado como evento relevante.

Finalmente, las relaciones de disparo como mecanismo adicional de interacción entre objetos son leyes interobjetuales asociadas a objetos que pueden actuar como agentes, activando el evento correspondiente cuando la condición expresada en la ley se satisface.

2.3 Una aproximación a la formalización algebraica del modelo objetual

El objetivo de esta sección es sentar las bases de la formalización algebraica de los conceptos que conforman la propuesta de modelo objetual O^3 . Para ello presentaremos un sencillo marco formal algebraico en el que básicamente se va a dar una definición de clase, y se van a caracterizar los operadores que a partir de las clases predefinidas permiten construir clases complejas.

En el siguiente capítulo se presentará un Lenguaje de Especificación para Sistemas Abiertos y Activos (OASIS), que se ajusta a las definiciones que a continuación se van a dar. La completa caracterización algebraica del entorno de especificación OASIS dentro de un modelo O^3, que constituye la aportación final de la presente Tesis en lo que hace referencia a la modelización del modelo objetual, se presentará entonces.

2.3.1 Noción informal de clase

El entorno de prototipación para SI Abiertos y Pasivos MOL [23] [91] [93] [94] constituye el punto de partida en lo que se refiere a la noción de clase como un ente con una estructura algebraica compuesta por:

1) aspectos estructurales caracterizados mediante la definición de atributos.

2) aspectos de comportamiento, caracterizados básicamente a través del concepto de evento.[6]

   La vida de un objeto se representa como la secuencia de eventos formada por todos aquellos eventos que han acontecido durante su existencia, comenzando obviamente por el correspondiente evento de creación (propio de la clase), y finalizando (si el objeto ha dejado de existir) con el de destrucción.

[6] Veremos más tarde como otras nociones como precondiciones de eventos y disparos juegan también un papel importante en la caracterización del comportamiento de una clase.

3) Una función de observación que relaciona los aspectos estructurales con los aspectos de comportamiento. Mediante esta función se puede expresar el valor de los atributos que conforman la estructura de un objeto en un instante dado, en función de su comportamiento (su vida).

La definición de la estructura algebraica correspondiente a una clase tiene que incluir por tanto las siguientes componentes:

* Un conjunto de eventos, dividido en dos subconjuntos:

  1) **eventos propios**, que son los que participan en la vida (traza) de objetos de una única clase (aquélla a la que el evento pertenece como tal evento propio)

  2) **eventos compartidos**, que forman parte de las vidas (trazas) de objetos de diferentes clases (las clases que comparten el evento en cuestión)

  Una ocurrencia de evento afecta a uno o más objetos. Asociada al evento está la información de cuáles son estos objetos participantes, más cualquier otra información adicional propia del evento.

  Entre el conjunto de eventos hay dos elementos distinguidos, que son los eventos creadores ('new') y destructores ('destroy') de objetos de la clase.

* Un conjunto de atributos tipados. Todo atributo toma valores del tipo (dominio) asociado con él. Distinguimos dos clases de atributos:

  1) **atributos constantes**, cuyo valor no se modifica durante la vida del objeto (propiedades estáticas).

  2) **atributos variables**, cuyos valores cambian como consecuencia de la ocurrencia de eventos (propiedades dinámicas).

  Cada objeto de una clase tiene un nombre o clave que lo identifica unívocamente durante toda su existencia. Esta clave está formada por un atributo constante (o una combinación de ellos en el caso más general).

* Un conjunto de trazas o ciclos de vida, que son la representación de los objetos o instancias de una clase.

  Cada una de estas trazas empieza con el correspondiente evento creador, que asigna un identificador a la instancia creada y da valor a sus atributos constantes, y continua con la secuencia de eventos que conforman la vida del objeto. Esta secuencia no es aleatoria. Nuevos eventos formarán parte de la traza asociada a un objeto sólo si se cumplen las precondiciones asociadas a ese evento.

* Para relacionar trazas con atributos, debe ser definida una función de observación . Esta función permitirá efectuar observaciones de cualquier objeto del Sistema. Dada una traza que representa la vida de un objeto, la función de observación va a dar los valores de los atributos de dicho objeto.

  Como un atributo tiene un valor único en un instante determinado, no puede aparecer más de una vez al efectuar una observación. Consideraremos que una traza vacía denota un objeto inexistente. De esta forma, la observación de un objeto inexistente es siempre vacía.

Las clases de nuestra Sociedad de Objetos son definidas sobre **dominios** (clases primitivas). Los dominios constan de un conjunto de valores (soporte) y un conjunto de operaciones sobre tales valores. El tipo del dominio es el conjunto soporte del correspondiente Tipo Abstracto de Datos (TAD).

Los dominios denotan la subespecificación de datos para las clases y se usan como identificadores de objetos y como tipos asignados a los atributos. Sus instancias existen siempre, no son creadas ni destruidas y no cambian nunca de estado. Como ejemplos. tenemos los conocidos referentes a *Nat* y *Bool* con sus tipos triviales {0,1,2,...} y {true,false} respectivamente.

Si una clase tiene eventos y atributos, y no ha sido construida utilizando operadores de clase decimos que es una **clase elemental**. **Clases complejas** son aquéllas construidas haciendo uso de otras clases mediante algún operador de clase.

Un SO puede ser así descrito de forma incremental, por combinación de clases predefinidas. Los operadores de clase que se van a usar son los de agregación, especialización, asociación y composición paralela, ampliamente usados en el area de la Modelización Semántica [65]. Más tarde veremos cómo el SO en su conjunto es una clase compleja obtenida por composición paralela del resto de las clases componentes.

Veamos cómo formalizar en este contexto la noción de clase.

2.3.2 Definición formal de clase

Abordaremos en primer lugar la definición sintáctica de clase, y fijaremos a continuación su semántica.

Sintaxis

Dado un conjunto inicial de géneros ('sorts') S, vamos a elaborar una signatura heterogénea $\Sigma = (S, O)$, siendo S el conjunto de géneros y O un conjunto de operaciones.

Definición 2.7 *Dado un conjunto E de eventos, se define sobre E una función rango r_e como:*

$$r_e : E \rightarrow S^+$$

que caracteriza el género de los argumentos de un evento.

Dado un $e \in E$, su **aridad** *(ar_e) se define como la composición funcional de las funciones longitud (lt_e) y rango(r_e):*

$$ar_e : E \xrightarrow{r_e} S^+ \xrightarrow{lt_e} N$$

Definición 2.8 *Dado un conjunto A de atributos, se define sobre A una función rango r_a como:*

$$r_a : A \rightarrow S^2$$

que caracteriza el género de los argumentos de un atributo.

La aridad ar_a de todo atributo es 2.

$$\forall a \in A, ar_a(a) = 2$$

Definición 2.9 *Dado un conjunto E de eventos con dos elementos distinguidos new y destroy, y dada una familia A (indexada por S) de conjuntos de atributos* $\{A_s \mid s \in S\}$, *se define inductivamente una* **(E,new,destroy,A)-fórmula** *del siguiente modo:*

* S_i**-términos**

1) una constante de género $S_i \in S$ *es un* S_i*-término*

2) una variable de género $S_i \in S$ *es un* S_i*-término*

3) si $a \in A_{S_i}$ *(es decir, a es de género* S_i*), y* t_1, t_2 *son términos de los géneros correspondientes al rango de a, entonces* $a(t_1, t_2)$ *es un* S_i*-término*

4) Todo S_i*-término se construye usando exclusivamente las reglas anteriores.*

* **átomos.**

1) si $e \in E$, *e tiene aridad n* $(ar_e(e) = n)$ *y* $t_1, \ldots t_n$ *son términos de los tipos correspondientes al rango de e, entonces* $e(t_1, \ldots, t_n)$ *es un átomo*

2) Si a,b son S_i*-términos, entonces* $a\Omega b$ *es un átomo, siendo* Ω *un operador del conjunto de operadores relacionales definidos para* S_i.

3) Sólo podemos obtener átomos por aplicación de las reglas anteriores

* **fórmulas.**

1) Un átomo es una fórmula.

2) Si A, B *son fórmulas, entonces* $A\&B, A \mid B, \not A$ *son fórmulas ('and', 'or' y 'not' lógicos resp.)*

3) Sólo obtenemos fórmulas a partir de las dos reglas anteriores. □

Definición 2.10 *Una* **precondición** *es una (E,new,destroy,A)-fórmula.*□

Definición 2.11 *Dado un conjunto E de eventos y un conjunto P de precondiciones, se define la función Pc:*

$$Pc : E \rightarrow P$$

Esta función Pc asocia a todo evento la precondición que debe de satisfacerse para que el evento sea relevante.

Definición 2.12 *Una* **clase** *es una 7-tupla*

$$C = (E, new, destroy, r, A, T, \alpha)$$

donde:

* *E es un conjunto de eventos con su aridad correspondiente y con dos elementos distinguidos new y destroy, que representan a los eventos creadores y destructores de instancias que toda clase tiene que poseer.*

* *r es la función rango definida sobre E, que caracteriza el género de los argumentos de un evento:*

$$r : E \rightarrow S^+$$

* *A es una familia indexada por S de conjuntos de atributos. A cada atributo se le asigna su correspondiente género por medio de una función 'sort':*

$$sort : A \rightarrow S$$

* *T es el conjunto de trazas o ciclos de vida. Cada traza está compuesta por la secuencia de eventos que representan la vida de un objeto.*

  Formalmente, $T \subset E^*$ y todo elemento de $x \in T$ (toda traza) satisface dos condiciones:

  1) $\forall x \in T \mid x = x_0 \bullet \ldots \bullet x_n$, se cumple que $x_0 = new$

2) *Cada evento tiene asociada una* **precondición**, *de forma que dado un evento* $e \in E$ *y una traza* $x \in T, (x = x_0 \bullet \ldots \bullet x_n)$, *obtendremos una nueva traza* x' *por concatenación del evento* e *a la traza de partida* x $(x' = x \bullet e)$ *si la precondición correspondiente* $(Pc(e))$ *se evalúa a cierto en el estado del objeto representado por la traza* x.

* α *es la función de observación, definida como:*

$$\alpha : T \rightarrow obs(A)$$

siendo $obs(A)=P_f(\cup_{a \in A}\{a\} \times \{sort(a)\})$, *es decir, el conjunto de las partes finitas de pares atributo-valor.*

Con α *obtendremos para cada objeto representado por medio de su traza, su correspondiente conjunto de pares atributo/valor como una función de la secuencia de eventos que le han sucedido.*□

La definición de clase induce una partición $O = O_{consultor} \cup O_{constructor}$ sobre el conjunto O de operaciones de la signatura Σ. El conjunto $O_{constructor}$ está formado por los eventos de la clase incluyendo *new*, *destroy*, y el conjunto $O_{consultor}$ está formado por sus atributos. La función de observación α se describe por medio de un conjunto de axiomas que definen las operaciones no constructoras en función de las constructoras.

Vamos a ver ahora cuál es la semántica asociada a la formalización sintáctica presentada.

Semántica asociada

La Semántica asociada a una clase C es una $\Sigma - Algebra\mathcal{A}$, es decir, una cierta interpretación de la signatura Σ correspondiente a la clase C. Tenemos pues que:

$$C \xrightarrow{i} \mathcal{A}$$

Se ha visto en el apartado anterior que el conjunto de operaciones de Σ era la unión de constructores (eventos) y consultores (atributos). Veamos cómo fijar esa interpretación i sobre las componentes relevantes de la definición de una clase, tomando como dominio de la interpretación el Universo de Herbrand (heterogéneo) sobre la Signatura Σ.

Introducimos en primer lugar la noción de **prefijo** de una traza, necesaria más adelante:

Definición 2.13 *Dada cualquier traza t, $t = t_1 \bullet \ldots \bullet t_n$ se define su k-prefijo $\wp_k(t)$ inductivamente del siguiente modo:*

* $\wp_1(t) = nil$

* $\wp_i(t) = t_1 \bullet \ldots \bullet t_{i-1} \quad \forall i : 2 \ldots n$

Definición 2.14 *La interpretación i que caracteriza la semántica de una clase C a partir de una signatura Σ es $i = (i_E, i_A, i_T, i_o)$ donde:*

* $\forall e \in E, e \xrightarrow{i_E} f_e$ *con* $f_e \in \mid O_{constructor} \mid$. i_E *es una función inyectiva.*

  Los eventos son los constructores de términos. En el contexto del modelo O^3, los eventos son los operadores generadores de términos que representan objetos de una clase. Estos términos van a ser usados para deducir los valores de los atributos en cualquier instante. Cumplen que:

  1) $sort(f_e)$=C

  2) $new_C :\rightarrow C$

* $\forall a \in A, a \xrightarrow{i_A} f_a$ *con* $f_a \in \mid O_{consultor} \mid$. i_A *es nuevamente una función inyectiva.*

  Los atributos se representan como operadores consultores de la signatura Σ

* $T \xrightarrow{i_T} \{\eta \mid \eta \in Fr_{\Sigma_{constructor}}(\emptyset)$ *(el álgebra libre sin variables (sobre el conjunto vacío)), tal que si* $t \in T$ *y* $(t{=}x_1 \bullet \ldots \bullet x_n)$ *y* $\wp_k$ *es*

el k-prefijo de t ($\wp_k = x_1 \bullet \ldots \bullet x_{k-1}$), *entonces* $\models_{\wp_k} Pc(x_k)$ $\forall k : 1 \ldots n$.[7]

Intuitivamente, una traza (compuesta por una secuencia de eventos relevantes, entendiendo por relevante que satisfacen su correspondiente precondición en el estado en el que acontecen) se interpreta como un término η *del álgebra libre sin variables usando los operadores constructores de la signatura (que son los eventos, como acabamos de establecer).*

* *En la sección anterior se ha definido* $\alpha : T \rightarrow P_f \mid A \times sort(A) \mid$. *La interpretación* i_α *correspondiente será:*

$$i_\alpha : \alpha \rightarrow (\Pi_{a \in A} T_a \rightarrow a \times \mid sort(a) \mid)$$

siendo $T_a \subset Fr_\Sigma(\emptyset)$ *la proyección de* T *sobre los eventos relevantes a* a, Π *el producto cartesiano de funciones (una por atributo* a*) que llevan* T_a *sobre los pares* $a \times \mid sort(a) \mid$, *y siendo* $\mid sort(a) \mid$ *el soporte de sort(a) (que representa el conjunto de valores que puede tomar el atributo considerado).*

i_α *viene caracterizada por ese producto cartesiano de funciones que para cada atributo, toman la proyección de una traza sobre los eventos que afectan al valor del atributo, y devuelven el término constante que representa su valor. Dichas funciones se definen axiomáticamente.* □

[7] De la traza $t \in E^*$, con $t = x_1 \bullet \ldots \bullet x_n$ pasamos al término $\eta \in Fr_{\Sigma_{constructor}}(\emptyset)$ concatenando por la derecha en el rango de x_i el género C cuyo nombre es el de la clase a la que pertenece el evento, y haciendo que el género de todos los eventos sea dicho género C. Al tratarse de dos notaciones homomorfas, las usaremos indistintamente en lo que sigue. Es decir:

$$\text{si } t = x_1 \bullet \ldots \bullet x_n \text{ y } r(x_i) = S_{i1}, \ldots, S_{in} \; \forall i : 1 \ldots n,$$

entonces

$$\eta = \overline{x}_n(\ldots \overline{x}_1(\ldots)\ldots), \text{ con } r(\overline{x}_i) = r(x_i) \bullet C \text{ y } sort(\overline{x}_i) = C$$

2.3.3 Caracterización de Operadores entre clases

Presentada la noción formal de clase, O^3 tiene que proporcionar mecanismos constructivos que permitan definir clases complejas a partir de las clases predefinidas. En este contexto algebraico esos mecanismos constructivos van a ser operadores entre clases. Una clase compleja es también una clase (es decir, una 7-tupla), con algunas restricciones dependientes del mecanismo (operador) utilizado para definirla.

Los operadores predefinidos[8] van a ser los conocidos mecanismos de abstracción usados en el area de la Modelización Semántica: agregación, especialización y asociación, con sus correspondientes inversos (respectivamente, proyección, generalización y pertenencia), usados en este contexto como mecanismos de organización de clases en un mismo nivel de abstracción de acuerdo con algunas propiedades dadas. Se respeta así la esencia de la propuesta FRISCO a ese respecto.

A ellos añadiremos el de composición paralela, que nos permitirá definir en última instancia una Sociedad de Objetos como una clase compleja obtenida por Composición Paralela de todas las clases componentes. De esta forma se dispone de un mecanismo constructivo simple y potente para especificar SO.

Desde la perspectiva clásica utilizada en los Modelos Semánticos, la agregación, especialización y asociación sólo abordan aspectos estructurales estáticos del SO que se modela. La aportación de esta propuesta es la de generalizarlos con aspectos dinámicos, dentro de un contexto OO. Para ello vamos a caracterizar la 7-tupla correspondiente a una clase compleja obtenida por aplicación de cada uno de dichos operadores. Como en la 7-tupla se encapsulan aspectos estructurales y de comportamiento, estaremos precisando así qué propiedades estáticas y dinámicas tiene una clase compleja en función de cómo (con qué operador de clases) ha sido construida.

Para clarificar la definición de estos operadores, se hace uso de un ejemplo sencillo que se va a refinar sucesivamente. Se trata de un SO constituido por una biblioteca, en la que inicialmente sólo consideramos

[8]Ese conjunto de operadores puede ser extendido con otros operadores, caracterizando esos nuevos operadores de acuerdo con la definición de clase, de forma análoga a como se va a hacer a continuación con los operadores citados. Se mantiene siempre el principio de que toda clase compleja es una clase más del Sistema.

la existencia de libros y socios, relacionados mediante préstamos con los que se autoriza a un determinado socio a sacar de la biblioteca un libro.

Agregación/Proyección

El operador **agregación** combina clases componentes, y genera una nueva clase con propiedades emergentes (propias de la clase agregada como nueva clase) y heredadas (de las clases participantes en la agregación).

Entre las propiedades emergentes están los nuevos eventos y atributos que sean definidos al agregar, mientras que son propiedades heredadas los eventos compartidos entre las clases componentes, vistos ahora por la clase agregada resultante como propios. Asimismo, son heredados los atributos constantes clave (identificadores) de las clases componentes.

Formalmente, podemos definir la agregación en términos de la 7-tupla de la siguiente forma:

Definición 2.15 *Dadas dos clases* C_1, C_2, *con*

$$C_1 = \{E_1, new_1, destroy_1, r_1, A_1, T_1, \alpha_1\}$$

y

$$C_2 = \{E_2, new_2, destroy_2, r_2, A_2, T_2, \alpha_2\},$$

se define su **agregación** *como una clase*

$$C = (E, new, destroy, r, A, T, \alpha)$$

en la que:

* $E = E' \cup E"$, *siendo*

  1) $E' = \{e \mid e \in E_1 \cap E_2 \;\; y \neg\exists\, C_3 = \{E_3, new_3, destroy_3, r_3, A_3, T_3, \alpha_3\} \mid e \in E_3\}$
  Intuitivamente, la intersección de E_1, E_2 *la componen los eventos compartidos entre ambas clases. Estos eventos son eventos propios de la clase compleja agregada.*

2) E" un conjunto de nuevos eventos definidos en la clase agregada (propiedades emergentes), tales que $E" \cap E_1 = \emptyset$ *y* $E" \cap E_2 = \emptyset$

* $new \in E'$

  El evento generador de instancias de la agregación es uno de los eventos compartidos entre las clases componentes.

* $destroy \in E'$

  El evento destructor de instancias de la agregación es también uno de los eventos compartidos entre C_1 *y* C_2.

* *La función rango r asociada a cada evento se define de forma natural. Obviamente, si* $e \in E$ *es un evento compartido, los rangos de e en C, C1 y C2 sólo diferirán en la última componente que referencia la clase que se define. Es decir,*

  - $r_1(e) = S_1 \ldots S_n \times C_1$
  - $r_2(e) = S_1 \ldots S_n \times C_2$
  - $r(e) = S_1 \ldots S_n \times C$

* *El conjunto A de atributos está compuesto por el conjunto de propiedades emergentes de C, más los atributos constantes identificadores de los objetos componentes.*

* *El conjunto de trazas (ciclos de vida) T viene dado en la forma habitual a partir de los eventos de la clase (E), con las siguientes restricciones adicionales (que proceden del hecho de que se trata de una clase compleja):*

  - $\forall t \in T, \exists t_i \in T_i, i = 1,2$ *tal que la proyección de* t_i *sobre aquellos de sus eventos compartidos que son eventos propios de C, es igual a la proyección de t sobre dichos eventos.*
  - *Y la condición opuesta:* $\forall t_i \in T_i$ *con i=1,2,* $\exists t \in T$ *tal que la proyección de t sobre aquellos de sus eventos privados que son compartidos con* C_i*, es igual a la proyección de* t_i *sobre esos mismos eventos.*

Estas dos condiciones imposibilitan la existencia de situaciones en las que un evento e de una clase C, que sea a la vez un evento compartido de dos clases C_1, C_2 pueda ocurrir (ser relevante) sin satisfacer las precondiciones establecidas en cada una de dichas clases.

* *La función de observación α no está sujeta a ninguna restricción adicional respecto a las funciones de observación $\alpha 1, \alpha 2$ de las clases componentes de la agregación. Se trata por lo tanto de una propiedad emergente.*

Vamos a usar como ejemplo el caso de una biblioteca muy elemental, compuesta por un conjunto de libros, socios y préstamos de libros a socios. *Libros* y *Socios* son clases elementales, mientras que los préstamos se representan como una clase compleja **préstamo** definida como la agregación de **libro** y **socio**. Esto implica que:

1) Existen eventos compartidos entre libro y socio. En concreto, estos eventos son **prestar** (un libro a un socio) y **devolver** (un socio devuelve un libro).

2) **Prestar** hace las veces de evento generador de instancias de la agregación, creando objetos de la clase *Préstamo*.

3) **Devolver** actúa como evento destructor de instancias de la clase préstamo, entendiendo por destructor que el intervalo de vida asociado a la instancia afectada queda cerrado[9].

4) La clase **Préstamo** tiene sus propios atributos (propiedades emergentes) como la **fecha del préstamo** o el **número asignado**, y tiene como atributos adicionales los que referencian a sus componentes: **libro** y **socio** en el ejemplo.

La definición de agregación se generaliza de forma trivial al caso en el que las componentes de la agregación son más de dos.

[9]Al hacer uso de un estilo deductivo o temporal, entendemos por evento destructor aquél que deja el objeto en un estado de no existencia sin perder toda su historia pasada. Estos eventos se añaden a la traza que representa la vida del objeto, como cualquier otro evento, y constituyen la última componente de dicha traza.

Una vez generada una clase agregada, se puede definir un operador **proyección** que devuelve la componente n-ésima de la agregación en cuestión. Cualquier componente es accesible a través del atributo constante inducido en toda agregación por sus clases componentes.

Especialización/Generalización

Los operadores de Especialización/Generalización dan soporte a la noción de herencia, fundamental en todo entorno OO. Es mucho el trabajo realizado en los últimos años en cuanto a herencia y OO se refiere. La herencia en este contexto es un mecanismo potente y extendido para la compartición de código y comportamiento entre clases, que posibilita la reutilización de Software y se manifiesta habitualmente en forma de estructura de grafo (para incluir tanto herencia simple como múltiple), heredando los hijos o descendientes propiedades definidas en sus respectivos padres.

Los modelos semánticos han ido incluyendo en sus propuestas progresivamente tales mecanismos en forma de relaciones IS-A, hasta el punto que no se concibe en la actualidad modelo semántico alguno que no soporte tal expresividad.

En O^3, las redes de herencia se van a representar en forma de operador entre clases. Nuevamente, la caracterización formal de la noción de herencia se traduce en la definición de la 7-tupla obtenida por aplicación del operador de especialización.

Definición 2.16 *Dada una clase* C_1, *con*

$$C_1 = \{E_1, new_1, destroy_1, r_1, A_1, T_1, \alpha_1\}$$

se define la **especialización** *de* C_1 *como la clase* C,

$$C = \{E, new, destroy, r, A, T, \alpha\}$$

tal que:

* $E = E_1 \cup E'$, *siendo* E' *un conjunto de nuevos eventos definidos en* $C1$ *(propiedades emergentes)*

* *El origen de new permite diferenciar dos situaciones:*

1) *Si la clase especializada posee su propio evento de creación de instancias, es decir, si dicho evento* $new_1 \neq new \in E$, *se tiene una* **especialización temporal**. *El objeto especializado sólo existe una vez ocurrido este evento, que es una propiedad emergente de la clase compleja especializada.*

2) *Si* $new_1 = new \in E$ *hablamos de* **especialización universal**, *entendiendo por universal que el objeto especializado está ligado al objeto padre durante toda la existencia del padre. En este tipo de especialización, hay que especificar la condición que caracteriza a los objetos de la clase especializada.* [10]

* *La situación con respecto a destroy es dependiente de lo comentado en el punto anterior. Si la especialización es temporal, destroy es una propiedad emergente. En caso contrario,* $destroy_1 = destroy$

* *r se define de la forma habitual, con la restricción obvia de que si un evento* $e \in E$ *y* $e \in E_1$ *entonces* $r_1(e)$ *y* $r(e)$ *sólo diferirán en la última componente, que referencia la clase* C_1 *y* C *respectivamente.*

* $A = A_1 \cup A'$ *siendo A' un conjunto de atributos propios de la clase especializada (propiedades emergentes).*

La persistencia o no persistencia del identificador de la clase padre en la clase especializada caracteriza dos situaciones diferentes:

1) *Existe* **dependencia de identificación** *cuando el identificador es el mismo en la clase padre e hijo.*

2) *Cuando la clase especializada se identifica independientemente del padre, entonces se dice que existe* **independencia de identificación**.

[10] El concepto de especialización temporal es análogo al concepto de 'role' que aparece en [70], donde 'role' y especialización se presentan como dos abstracciones distintas (aunque muy próximas). El operador de especialización definido en O^3 unifica esas dos situaciones.

En cualquier caso, los eventos de la vida del hijo que son heredados del padre tienen que reflejarse necesariamente en la vida del padre y viceversa, como se establece a continuación.

* *T se define de la forma habitual. El comportamiento de la clase hija se puede especializar modificando las precondiciones asociadas a eventos de la clase base. De cualquier forma, dada una traza $t \in T$, y otra traza $t_1 \in T_1$,se verifica que la proyección de t sobre los eventos de la clase padre C_1 es igual a t_1*

* *α se define teniendo presente que las definiciones de los atributos de la clase especializada que son también atributos de la clase padre deberán ser las mismas:*

$$Si\ \Re = A \cap A_1,\ entonces\ \alpha\ |_{\Re} = \alpha_1\ |_{\Re},$$

aunque también podríamos especializar α enriqueciendo la definición axiomática de atributos comunes con propiedades emergentes de la clase especializada. En cualquier caso, la restricción anterior evita inconsistencias en el caso en que propiedades emergentes (p.e. eventos) de la clase especializada modifiquen propiedades de la clase padre (p.e. atributos)[11] □

Veamos algunos ejemplos en el caso práctico de la Biblioteca. Supongamos que normalmente cada socio puede tener prestados como máximo 5 libros simultáneamente, que un socio adquiere el status de 'socio poco fiable' cuando ha devuelto libros fuera de plazo en más de 10 ocasiones, y que en ese caso no se permite tener en préstamo más de 1 libro.

En ese caso, la clase 'Socio_no_fiable' es una especialización de la clase 'Socio', que hereda eventos y atributos de 'Socio', y que tendrá como propiedad emergente la de que sólo puede sacar un libro cada vez de la Biblioteca. Esta propiedad emergente se expresa a través de la precondición asociada al evento *prestar*, que no será la misma existente en la clase base 'Socio'.

[11] Al presentar el Lenguaje de Especificación OO Oasis se aportarán soluciones prácticas a todas estas situaciones.

El ejemplo es un caso claro de especialización temporal, pues sólo cuando por décima vez un socio devuelva un libro fuera de plazo, adquiere el status de no fiable.

Se tiene asimismo dependencia de identificación, pues asumimos que el atributo clave de los objetos de la clase 'Socio_no_fiable' es el mismo que el atributo clave de los objetos de la clase 'Socio'.

Caracterizemos ahora la operación opuesta a la especialización, que es la **generalización**.

La generalización se usa para elaborar una vista uniforme de diferentes clases base. Básicamente, se trata de la operación inversa a la especialización: con la especialización se define un número de subclases para una clase base dada mientras que en la generalización se define una superclase común a partir de un conjunto de subclases.

Definición 2.17 *Dadas dos clases* C_1, C_2, *con*

$$C_1 = \{E_1, new_1, destroy_1, r_1, A_1, T_1, \alpha_1\}$$

y

$$C_2 = \{E_2, new_2, destroy_2, r_2, A_2, T_2, \alpha_2\}$$

se define su generalización como la clase

$$C = \{E, new, destroy, r, A, T, \alpha\}$$

con las siguientes propiedades;

* $E = E_1 \cap E_2$, *es decir la clase generalizada tiene como eventos los elementos de la intersección de las clases que se generalizan.*

* *El evento generador de instancias de la clase generalizada (new) es una propiedad emergente. No puede ser de otro modo, pues si* $new_1 = new_2$, *entonces ese sería el evento new. Pero esa situación es inconsistente, pues los objetos de* C_1 *serían los mismos de* C_2.

  Una vez creada la clase generalizada, C_1 *y* C_2 *son especializaciones de C, por lo que es necesario estudiar si se ha realizado una generalización temporal o universal.*

Si la generalización es universal, los eventos new y sus new_i asociados deben sincronizarse. Esto se realiza renombrando los eventos new_i como new, y especificando en las nuevas clases especializadas la condición asociada a toda especialización universal.

Si la generalización es temporal no hay que establecer restricciones adicionales.

* *De acuerdo con lo expuesto en el punto anterior, destroy se define también como propiedad emergente.*

* *Como ocurre en la especialización, los rangos de C, C_1 y C_2 sólo difieren en su última componente, que referencia la clase considerada.*

* $A = A_1 \cap A_2$, *análogo al caso de* E.

  Con el identificador de la nueva clase generalizada se pueden presentar dos situaciones, que se corresponden a los casos opuestos a los que se tiene con la especialización.:

  - *si el identificador de la clase generalizada es el mismo de las clases 'hijas', tenemos generalización con dependencia de identificación. En este caso las claves de las clases que pasan a ser hijas de la clase generalizada son la misma.*
  - *si el identificador es una propiedad emergente, entonces la generalización tiene independencia de identificación.*

* T *se construye sobre* E, *de la forma habitual. Como en el caso de la especialización, cualquier ocurrencia de un evento* $e \in E$ *formará parte tanto de la traza de la clase generalizada como de la de sus clases hijas. Es decir, la proyección de una traza* $t_1 \in T_1$ *sobre los eventos de la clase generalizada debe ser igual a la traza* $t \in T$ *que representa al mismo objeto.*

* $\alpha\,|_{a \in A_1 \cap A_2} = \alpha_1\,|_{a \in A_1 \cap A_2} = \alpha 2\,|_{a \in A_1 \cap A_2}$. *Para que la operación de generalización sea consistente se asume que la definición de la función de observación es la misma para los atributos comunes. Esa exigencia es coherente con la noción de generalización.*□

En el contexto bibliotecario, supóngase que existen dos categorías diferentes de lectores: lectores_trabajadores y lectores_en_paro. Si asumimos que inicialmente se han definido como dos clases distintas y que tienen tanto propiedades en común (su DNI como identificador, un nombre, una dirección,...) como propiedades que los diferencian (los lectores_trabajadores pagan una cuota anual, los parados no, etc.), podemos definir la clase 'Lector' como generalización de las dos, de acuerdo con la definición dada:

* Los Eventos y Atributos de la clase *lector* son los comunes a las clases *lectores_trabajadores* y *lectores_en_paro*.

* Nuevos eventos *new* y *destroy* han de ser definidos para la clase *lector*. Como este ejemplo es un claro caso de especialización universal, se hace necesario sincronizar el evento creador de instancias de *lector* con la clase hija que corresponda. Al crear un lector, se está creando también un lector en paro o un lector trabajador.

* Las trazas o ciclos de vida de objetos de la clase *lector* van a ser proyecciones de las trazas de objetos de la clase especializada correspondiente sobre los eventos comunes.

Asociación/Pertenencia

El tercer operador de clases considerado es el de **asociación**. La asociación es introducida como mecanismo de abstracción en muchas propuestas recientes de Modelos Semánticos ([65] [19]), aunque la respuesta a la cuestión de si la asociación es realmente un mecanismo de abstracción no esté clara. ¿Es correcto interpretar un *departamento* como una abstracción de un conjunto de empleados?

La aproximación funcional en Modelización Semántica [119] no usa el concepto de asociación, sino que lo sustituye por atributos multivaluados vistos como funciones que asocian a un atributo de un objeto otro objeto de tipo colección (conjunto, lista, cola,...). Otras propuestas sí que hacen uso de un mecanismo de abstracción explícito de asociación ([58]).

Existen también aproximaciones que prescinden de la asociación, bajo el argumento que toda asociación es equivalente a un conjunto de agregaciones ([57]).

De cualquier forma, en el contexto del modelo OO y ontológico O^3 que se presenta en este trabajo, se va a utilizar la asociación como un operador de clases, que permita manipular de una forma sencilla y clara situaciones en las que una clase tiene como propiedad fundamental el estar relacionada con muchos objetos de otra, aunque posea otras propiedades que consideraremos emergentes de la clase asociada. Así, las instancias de una clase compleja generada por asociación contendrá una colección de instancias de la clase base. A estas instancias componentes se les puede dar una estructura ordenada (por medio de listas, pilas o colas) o desordenada (si se usa conjuntos (sin un orden asociado) o multiconjuntos ('bag's) como mecanismo de agrupación).

Llamaremos *miembros* de una clase asociada a un atributo cuyo valor para un objeto dado de la clase compleja devolverá la colección de instancias de la clase base asociadas a él.

Vamos a caracterizar la clase compleja asociación en términos de la 7-tupla. A la clase cuyas instancias se agrupan la vamos a llamar *clase base.*

Definición 2.18 *Dada una clase* C_1 *con*

$$C_1 = \{E_1, new_1, destroy_1, r_1, A_1, T_1, \alpha_1\}$$

se define su clase asociada C *como la 7-tupla*

$$C = \{E, new, destroy, r, A, T, \alpha\}$$

que satisface:

* $E \supset \{insert, delete\}$, *donde*

  1) insert es el evento de inserción de nuevos componentes de la clase base en el objeto considerado de la clase asociada

  2) delete es el evento de borrado de componentes existentes en una instancia de la clase asociada

La inserción y borrado de componentes puede ser automática o manual dependiendo de si existe o no existe respectivamente un criterio de agrupación (un cierto valor de atributo de la clase base, etc.).

Si el criterio existe, inserciones, modificaciones o borrados en la clase base originan automáticamente cambios en las componentes afectadas de la asociación. En este caso, una vez creada una instancia de la clase compleja, se necesita un atributo que dé valor al criterio de clasificación usado. Es necesario para asociar instancias de la clase compleja con el valor de ese atributo en objetos de la clase base. Como los valores de ese atributo son forzosamente distintos para toda instancia de la clase agrupada, siempre puede ser utilizado como clave de la misma.

* *new, destroy son propiedades emergentes de la clase asociación.*

* *r tampoco tiene ninguna restricción con respecto a* r_1

* $A \supset \{miembros\}$. miembros *es un atributo variable, cuyos eventos relevantes son los de inserción y borrado de componentes* (insert,delete). *Este atributo* miembros *es de un tipo genérico colección (conjunto, lista, pila o cola de X, siendo X el tipo de la clase base de la asociación)*

* *T se define sin restricciones con respecto a* T_1. *Obviamente, toda instancia introducida como componente en una asociación, debe existir.*

* α *se define asimismo sin restricciones respecto de* α_1, *incluyendo la definición del nuevo atributo variable* miembros *en función de sus eventos relevantes* insert,delete. □

Para ilustrar el operador asociación con un ejemplo, supongamos que los libros de la biblioteca están distribuidos por descriptor temático fundamental *microthesaurus*, y que cada grupo tiene propiedades propias como su *ubicación* en una u otra sala, o el número de *días* que puede ser prestado.

Se define en esa caso una clase compleja *libros_por_temas*, como una asociación de libros, con un criterio de agrupación: que su atributo *tema* sea el mismo.

En este ejemplo, la inserción y borrado de componentes es automática, ya que insertado un libro, formará parte de aquella instancia de clase compleja cuyo valor de *tema* sea el mismo. *Tema* debe ser como se indicaba antes un atributo de la asociación, que en este caso podría actuar como identificador de instancias de la clase compleja.

La pertenencia de un objeto de la clase base a una instancia de la clase compleja asociada se define mediante la operación In, asociada a la clase primitiva *colección* (set,list etc.) utilizada en la asociación:

$$In : C_1 \times C \rightarrow Bool$$

que, dadas dos instancias $c_1 \in C_1$, $c \in C$; devuelve el valor $true$ si c_1 está incluida en el valor del atributo $miembros$ de C, y $false$ en caso contrario. Es decir:

$$in(c_1, c) = true\ if\ c_1 \in miembros(c)\ otherwise\ false$$

Composición Paralela

La definición de este operador es otra aportación de O^3, ortogonal en su concepción a las aportaciones de los otros operadores. La Composición Paralela no proviene del area de la Modelización Semántica, sino que procede de la Teoría de Procesos [63] [86]. Al definir la agregación, la especialización y la asociación hemos partido de sus propiedades como mecanismos estructurales, para enriquecerlas con aspectos de comportamiento (dinámicos). En el caso de la composición paralela, el planteamiento es el inverso: partimos de un operador asociado a conceptos dinámicos y a la noción de proceso, y vamos a enriquecerlo con aspectos estrucuturales, en el contexto OO fijado por la definición de clase propuesta.

El operador de Composición Paralela va a permitir definir un SO como una Clase Compleja resultante de la composición paralela de las Clases definidas previamente. Dadas dos clases C_1, C_2, una instancia de la clase C Composición Paralela de C_1 y C_2 denota un elemento del

conjunto de las partes de la unión de los objetos de las clases componentes. Por tanto, si C_1 tiene como atributo clave a_1 de género t_1, y C_2 tiene como atributo clave a_2 de género t_2, entonces instancias posibles de C son:

* o_1 de género t_1 (escribimos $(o_1 : t_1)$)
* $\{o_1 : t_1, o_2 : t_2\}$
* $\{o_1 : t_1, o_1' : t_1, o_2' : t_2\}$
* ...

Definición 2.19 *Dadas dos clases* C_1, C_2, *con*

$$C_1 = \{E_1, new_1, destroy_1, r_1, A_1, T_1, \alpha_1\}$$

y

$$C_2 = \{E_2, new_2, destroy_2, r_2, A_2, T_2, \alpha_2\}$$

se define su Composición Paralela $C = C_1 \| C_2$ *como la clase*

$$C = \{E, new, destroy, r, A, T, \alpha\}$$

con las siguientes propiedades:

* $E = E_1 \cup E_2 \cup E_c$

  Intuitivamente, el conjunto de eventos de la composición paralela está formado por la unión de los conjuntos de eventos de las clases componentes, más el conjunto E_c *de eventos declarados explícitamente en la clase* C *(propiedades emergentes).*

* $new, destroy$ *son propiedades emergentes de la nueva clase compleja.*

* $\forall e \in E$, *si* $e \in E_i$ *y* $r_i(e)$ *es su rango, entonces* $r(e) = r(e_i) \times s_c$ *donde* s_c *es el género que representa a la clase* C.

  La idea es que todos los eventos de la composición paralela de dos clases añaden al rango del evento heredado de las clases componentes, un nuevo género que es el género que representa a la clase compuesta.

* $A = A_1 \cup A_2 \cup A_c$

  El conjunto de atributos de la clase compuesta está formado por la unión de los atributos de las clases componentes, etiquetados con la clave de la composición paralela, más el conjunto de atributos declarados en C como propiedades emergentes.

* $T = \{t \mid t \in E^*\}$.

  Cada traza del conjunto T de trazas o ciclos de vida de la clase compleja puede verse como el entrelazado de las trazas de las clases componentes, sincronizadas en sus eventos compartidos.

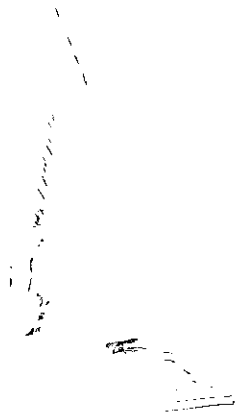
* *La función de observación* $\alpha = \alpha_1 + \alpha_2 + \alpha_c$, *es decir es la suma de las funciones de observación componentes (modificadas, ya que los eventos son distintos al haber añadido el género que representa a la clase compuesta) y la función de observación correspondiente a las propiedades emergentes de la clase compleja C.*

En el contexto de la biblioteca, usando el operador de Composición Paralela se define de forma natural el conjunto del SO Biblioteca como la Composición Paralela de las clases *Socio*, *Libro* y *Préstamos*. De esta forma diferentes instancias de la clase *Biblioteca* se corresponden con diferentes bibliotecas, compuestas cada una de ellas por un conjunto de libros, socios y préstamos acontecidos, representadas con su traza o ciclo de vida correspondiente.

Caracterizados los operadores de clases en O^3, y antes de presentar las Teorías Formales de Primer Orden correspondientes al entorno de producción de Software O^3, se presenta en la siguiente sección un Lenguaje de Especificación OO para SO Abiertos y Activos llamado OASIS ('Open and Active Specification of IS'), que recoge las nociones básicas del modelo O^3 y constituye la herramienta formal de especificación del entorno de producción de Software que se está desarrollando.

Capítulo 3

Oasis:Un Lenguaje de Especificación OO



Los Sistemas Organizacionales (SO) que queremos diseñar y desarrollar pueden ser grandes y complejos. Por esta razón, necesitamos herramientas que hagan posible su especificación. Este diseño comienza con la captación y representación de conocimiento sobre el SO objeto de estudio, incluyendo aspectos dinámicos y estáticos. A este nivel es irrelevante el indicar de qué forma va a ser implementado ese conocimiento, por lo que una herramienta de modelización debe proporcionar especificaciones **declarativas**. Como ocurre en cualquier tipo de ingeniería, el proceso de diseño debe dar como resultado modelos de soluciones sobre las que poder reflexionar antes de abordar la implementación concreta del Sistema.

Especificaciones formales de modelos deben obtenerse por tanto lo antes posible en el proceso de diseño de cualquier SO. Una característica deseable para cualquier Lenguaje de Especificación es que disponga de una representación lógica que permita razonar sobre las especificaciones escritas en el lenguaje dentro de un marco lógico, basado en la teoría de la demostración. Así se hace posible deducir conocimiento de las especificaciones con el fin de justificar posteriores decisiones en el proceso de producción de Software y posibilitar una verdadera auditoría informática verificando implementaciones sobre la especificación origen.

De acuerdo con todo lo anterior, ha llegado el momento de introducir un lenguaje de especificación que permite describir Sociedades de Objetos interactivos, en el contexto del modelo O^3 que se ha definido.

Vamos a utilizar el criterio de clasificación de Lenguajes de Especificación introducido en el capítulo 1, basándonos ahora en un modelo OO bien definido. Dicho criterio usa dos dimensiones ortogonales:

* **OO o no OO**, en función de su grado de afinidad con los principios de O^3

* **Deductivos (temporales) o Dinámicos**. [22]

  Un SO Deductivo es aquél en el que las actividades[1] que suceden se almacenan en el Sistema, y cualquier otro conocimiento del

[1] Utilizaremos la noción de actividad propuesta por FRISCO, dónde una actividad es un proceso discreto que permite distinguir el estado del Sistema antes y después de su realización.

En el modelo O^3 propuesto, las actividades se corresponden con eventos

mismo se deduce a partir de esa información.

Un SO Dinámico es aquél en el que en lugar de almacenar las actividades como Base de Información, ésta está compuesta por hechos básicos modificados convenientemente cuando una actividad ocurre en el Sistema (sólo se almacena el 'ahora').

Si se toma una aproximación dinámica, cuando se especifica una operación hay que conocer cuáles van a ser sus efectos sobre la información almacenada en el Sistema. En términos de nuestro modelo OO, la especificación de un evento tendría que conocer qué modificaciones se inducen en qué atributos como consecuencia de la ocurrencia de dicho evento.

En la aproximación temporal o deductiva, al definir las reglas de derivación hay que conocer todos los eventos externos (operaciones o transacciones) susceptibles de afectar a dichas reglas. En nuestro modelo OO, se trata de localizar en la definición de los atributos los efectos de los eventos que los modifican, con un estilo deductivo.

La utilización de una aproximación deductiva tiene una serie de ventajas muy deseables para el Diseño e Implementación de SO, entre las que destacan ([22]):

1) Favorece el conocimiento de los estados y del comportamiento del Sistema considerado.

2) Su dimensión temporal facilita el tratamiento de información histórica, frente a la instantánea del Sistema proporcionada en una aproximación dinámica.

3) Posibilita la generación de prototipos con propiedades formales bien caracterizadas, como vamos a ver en esta Tesis.

4) Su dimensión temporal y su estilo declarativo hacen menos difícil la introducción de cambios y la adecuación de nuevos requerimientos.

Según esta clasificación, el lenguaje de especificación que queremos desarrollar ha de cubrir el cuadrante OO y deductivo:

* OO para aprovechar todas las ventajas derivadas del uso de un modelo objetual a nivel de especificación.

* deductivo para dar un soporte formal lógico a lo especificado, en la línea que se ha comentado en los párrafos anteriores, sin ocuparse de aspectos de implementación irrelevantes a este nivel de abstracción.

El Lenguaje de Especificación de Sistemas de Información Abiertos y Activos **Oasis** [95] que presentamos en este capítulo recoge todos los conceptos expuestos anteriormente con el fin de convertirse en la herramienta expresiva necesaria para describir los Sistemas de Información en los que estamos interesados. Históricamente, es la evolución activa de los lenguajes de especificación del entorno de prototipación de SO pasivos MOL [23,24,91,93,94]. Estos lenguajes heredan el concepto de objeto desarrollado en [116] y están basados en el uso de un modelo OO, enriquecido con una expresividad lógica clausal y/o ecuacional.

3.1 Especificación de Clases en Oasis

Una especificación en Oasis se construye de acuerdo con el siguiente esquema:

Conceptual Schema nombre
    cuerpo de la especificación
end Conceptual Schema

El cuerpo de la especificación se realiza a partir de las tres unidades de diseño que proporciona Oasis:

* Dominios o Clases primitivas
* Clases Elementales
* Clases Complejas

3.1.1 Dominios o Clases Primitivas

Los dominios constituyen la mínima unidad de diseño y denotan la subespecificación de datos. Se usan como identificadores de objetos y como clases asignadas a atributos. La Sociedad de Objetos se construye

a partir de ellos, ya que son tomados como tipos de datos básicos sobre los que las clases elementales van a ser declaradas.

En el contexto de O^3 con los dominios se representan esas propiedades básicas asociadas a un objeto que rompen la circularidad de la noción tradicional de objeto[2]. Los dominios nos dan un conjunto de entidades "platónicas", que no cambian de estado y que existen eternamente.

Su tipo es un TAD, y algunos están predefinidos (otros pueden ser construidos por el diseñador). La versión actual de Oasis proporciona como dominios predefinidos nat, bool y string con sus operaciones habituales.

El diseñador puede enriquecer este conjunto de TAD's predefinidos con otros que necesite como soporte de la especificación. En particular, puede usar tipos genéricos para agrupar TAD básicos mediante los conocidos mecanismos de colección (conjuntos, pilas, colas etc.).

El tiempo es considerado también un dominio, implícitamente definido y necesario para aplicar el Cálculo Simplificado de Eventos de Kowalski al Lenguaje [75]. Va a aparecer como último argumento de los operadores que representen eventos y atributos, fijando respectivamente el instante en que un evento ocurre o el instante en que se desea realizar una observación (conocer los valores de los atributos de algún objeto). La representación del tiempo es discreta y acotada por la izquierda, e isomorfa a los números naturales.

Desde el punto de vista sintáctico, los dominios usados en una especificación se declaran al principio con la siguiente sintaxis:

domains nat,bool,string,time

3.1.2 Clases Elementales

Las Clases Elementales son aquéllas construídas sin necesidad de utilizar operadores de clase. Caracterizan las propiedades estructurales y de comportamiento que una colección de objetos (el tipo) comparten.

Veamos cuál es la estructura de una clase elemental. Sintácticamente consta de las siguientes secciones:

[2]En Ontología, un objeto ha de poder tener más de un estado sin perder su identidad. Los dominios (datos) no son pues objetos propiamente.

elementary class nombre
attributes
constant
nombre atributo:clase [**key**]
...
variable
nombre-atributo(nombre-clase-elemental,nombre-dominio
,tiempo) (valor-defecto)
fórmulas def-var
definición axiomática
...
static constraints
condición-admisibilidad-estática
events
private
nombre-evento(nombre-clase-elemental,[argumentos],
tiempo) [**new**] [**destroy**]
...
shared
nombre-evento(nombres-clases,[argumentos],tiempo)
...
preconditions
nombre-evento **if** condición-admisibilidad
...
triggering
destino::nombre-evento(nombres-clases,[argumentos],tiempo)
if condición-disparo
...
end elementary class

Se componen de las siguientes partes:

1) el nombre de la clase

2) un conjunto de atributos constantes (aquéllos cuyo valor no cambia durante la existencia de un objeto de la clase) y variables (caso contrario). Como todo atributo está clasificado, en la declaración

se indica el tipo asociado. Para las clases elementales, dicho tipo puede ser una clase primitiva (dominio) o cualquier otra clase de la especificación, dando soporte de forma natural a la noción de atributo objeto-valuado.

Entre los atributos constantes, se elige el identificador o clave, marcado con la palabra **key**. La clave puede ser compuesta, lo que se traduce en que todos los atributos constantes que la componen aparecen marcados con la palabra clave 'key'.

Al declarar un atributo variable se indica cuál es el valor que va a tomar por defecto al crear un objeto de la clase. Dicho valor ha de ser una constante del tipo del atributo, especificada entre paréntesis e indicada en el esquema anterior como *valor-defecto*[3]. Para cada atributo variable hay que especificar de forma declarativa su definición en función de los eventos que le son relevantes. Esta definición constituye la especificación en Oasis de la función de observación, ya que caracterizado un objeto a través de su traza (la secuencia de eventos que conforman su vida), la definición axiomática de los atributos del objeto en función de los eventos que pueden modificar sus valores, permitirá conocer en cualquier instante el valor de tales atributos.

En función de qué expresividad lógica se utilice en esta definición declarativa, se tendrán diferentes versiones de Oasis:

* Si la definición hace uso de una expresividad clausal, se trabaja en la versión clausal de Oasis (**C-Oasis**).
* Si la expresividad usada es la ecuacional, entonces estamos ante la versión funcional de Oasis (**F-Oasis**)
* Si se permite utilizar indistintamente una expresividad clausal o ecuacional, estamos ante la versión lógica (en sentido amplio) de Oasis (**L-Oasis**)

[3]Alternativamente, podría deducirse este valor del evento de creación correspondiente, de la misma forma que se define cualquier atributo variable. Ese evento de creación es relevante para todo atributo si se considera que actúa como inicializador. El uso del valor por defecto simplifica la notación.

Veamos con un ejemplo la distinta expresividad de cada una de las versiones del lenguaje. Supóngase que se quiere definir el atributo *número de libros* de la clase *socio*. Se trata de un caso claro de atributo variable, cuyos eventos relevantes son *prestar* y *devolver*. La acción de prestar un libro a un socio tiene como efecto el que el valor de su atributo *número de libros* se incrementa en uno. En el caso de la devolución, el efecto es el inverso y se debe restar uno al valor actual del atributo.

La definición en C-Oasis de este atributo usa como hemos dicho una expresividad clausal y se efectúa del siguiente modo:

```
num_libros(socio,nat,time) (0)
 clauses s:socio;n:nat;l:libro;t,t1:time;
  num_libros(s,n,t) if t1 is t-1, num_libros(s,n1,t1),
                       (prestar(l,s,t),n is n1+1;
                       devolver(l,s,t),n is n1-1;
                        n is n1).
```

La definición del atributo se realiza recursivamente en función del valor del atributo en el estado anterior, más la modificación inducida por sus eventos relevantes en el instante actual.

En la versión funcional, aunque el sustrato básico es el mismo, la definición de los atributos variables hace uso de una expresividad ecuacional. En el ejemplo,

```
num_libros(socio):nat (0) ;
 equations n:nat;s:socio;l:libro;t:time;
  num_libros(s)=n if T=T1'prestar(l,s,t) and
                  num_libros(s)=n-1 at T1.
  num_libros(s)=n if T=T1'devolver(l,s,t) and
                  num_libros(s)=n+1 at T1.
                  else num_libros(s) at T1.
```

Tendremos tantas ecuaciones como eventos relevantes tenga el atributo que se define. Es interesante resaltar lo siguiente:

(a) el uso de un lenguaje de trazas en la versión funcional. Este lenguaje referencia eventos componentes de trazas a través del operador de concatenación de eventos a la traza '. La fórmula $T = T1'e$ expresa que la traza T es el resultado de concatenar el evento e a la traza $T1$.

(b) Los atributos no llevan explícitamente una etiqueta temporal como en el caso clausal, pero sí de forma implícita, porque toda observación va asociada a una traza, que representa un estado particular del objeto observado, sobre la que puedo preguntar en cualquier instante de tiempo.

Así,cuando se escribe *num_libros(s)* nos estamos referiendo a *num_libros(s,T,t)*, siendo T una traza (la que representa el estado relevante) y t el instante considerado en el que se efectúa una observación.

(c) En la expresividad funcional el último axioma denota la regla marco, cuya interpretación es que si en el instante t no ha ocurrido ningún evento relevante para el atributo considerado, su valor será el que tenía en el instante t-1 anterior.

Por último, en L-Oasis la definición de atributos variables utiliza cláusulas de Horn con igualdad, donde el atributo y su valor aparecen en la cabeza de la cláusula, y el evento que lo modifica aparece en el cuerpo. Cada evento relevante para al atributo considerado da origen a una fórmula de la siguiente forma:

atributo(C,T)=valor :- evento(C,T)[4]

[4]valor se representa en el caso más general como una expresión que caracteriza el valor del atributo en el estado alcanzado al ocurrir el evento considerado, en un entorno deductivo: el valor del atributo en un estado dado se deduce a partir de la secuencia de eventos que constituye la vida del objeto, de acuerdo con esas fórmulas. El estado viene caracterizado por la variable T que denota una traza de eventos.

Las propiedades exigidas a las ecuaciones condicionales utilizadas en L-Oasis son las de 'level-confluencia'([85]). Esta condición permite definir una semántica operacional eficiente y completa para ejecutar el programa lógico- ecuacional equivalente a una Especificación L-Oasis (como veremos en el próximo capítulo), incluso en ausencia de la propiedad de terminación ('noetheriana'), o con variables extras en el cuerpo de las ecuaciones

donde:

(a) *atributo* será el nombre de un atributo.

(b) *valor* será la expresión que determina el valor que toma el atributo.

(c) *evento* será el nombre del evento que se está considerando relevante.

(d) C *y* T serán variables que representan una clase y un estado como una traza de eventos respectivamente.[5]

Con el ejemplo que venimos usando, se tiene la siguiente definición:

```
num_libros(socio,time):nat (0);
 formulas s:socio;l:libro;t:time;
  num_libros(s,t)=0 if inscribir(s,t).
  num_libros(s,t)=num_libros(s,t-1)+1 if prestar(l,s,t).
  num_libros(s,t)=num_libros(s,t-1)-1 if devolver(l,s,t).
```

3) Se declaran **restricciones de integridad estáticas** para indicar qué combinaciones de valores de atributos son consideradas válidas en todo estado. Estas restricciones están constituidas por conjunciones de átomos de un lenguaje de primer orden usando operadores relacionales para combinar términos construidos sobre nombres de atributos y constantes y variables de las clases de atributos.

Por ejemplo, la restricción estática que establece que un *socio* no puede tener más de 10 libros prestados simultáneamente se declararía como:

$$num\_libros(socio) \leq 10$$

[5] Estas variables de tipo "traza" van a ser consideradas variables de tipo "tiempo" en la especificación. Referirse a un instante determinado es equivalente en este contexto a referirse a un estado concreto caracterizado por una traza.

Estas restricciones pueden ser interobjetuales recogiendo las aportaciones al modelo O^3 de la propuesta formal de Wand referentes a leyes interobjetuales ([89]). Es decir, los atributos usados en los átomos del lenguaje de las restricciones estáticas pueden ser cualquier atributo de cualquier clase de la Sociedad de Objetos que se especifica. Aumenta así notablemente la capacidad expresiva del Lenguaje para describir situaciones en las que el valor de un atributo de una clase depende del valor de otro atributo de otra clase.

4) La declaración de los eventos de la clase tiene dos partes diferenciadas:

   (a) declaración de eventos propios. Los eventos propios se representan en C-Oasis como predicados[6] cuyo primer argumento es siempre el nombre de la clase propietaria, el último es el tiempo utilizado como etiqueta temporal, y entre ambos pueden haber otros argumentos que representen información adicional propia del evento.

   Los eventos generadores y destructores de instancias de una clase van etiquetados con las palabras especiales **new** y **destroy** respectivamente.

   (b) declaración de eventos compartidos. Los eventos compartidos tienen las mismas propiedades que los eventos propios, con la excepción de que, siendo más de una las clases propietarias del evento, todas aquellas clases que lo comparten aparecen como sus primeros argumentos.

5) La especificación de las precondiciones asociadas a los eventos de la clase da cuenta de cuándo un evento va a ser relevante. La condición de admisibilidad es una precondición tal como ésta se ha definido en la sección 2.3.2.

   Para aligerar la sintaxis de Oasis, se asume implícitamente definida siempre la existencia de los objetos participantes en un

[6] En F-Oasis y L-Oasis se representan, utilizando la notación funcional empleada en el capítulo anterior, como operadores cuyo género es la clase a la que pertenecen, y que incluyen como último argumento el de dicha clase.

evento como precondición por defecto. Esta existencia implícita se traduce en que el objeto debe haber sido creado en un instante anterior, y no haber sido destruido a partir de ese instante hasta el instante en que se activa el evento.

Análogamente, los eventos que crean y destruyen instancias de una clase tienen como precondición por defecto la no existencia (existencia) respectivamente de la instancia que se quiere crear (borrar).

La declaración de precondiciones en eventos compartidos puede conducir a una situación en la que la precondición global del evento, que ha de ser única, esté repartida entre las clases que lo comparten. A efectos de la especificación del SO, la precondición asociada a un evento compartido se define como la conjunción de las precondiciones declaradas para ese evento en cada una de las clases que lo comparten.

6) La parte final de la especificación de una clase la constituye la declaración de sus condiciones de disparo, especialmente importantes porque introducen actividad en el SO que se describe. Tal actividad se traduce en una visión del SO en la que ocurren eventos activados por los agentes del Sistema y eventos activados internamente por el propio Sistema cuando ciertas condiciones de disparo se satisfagan. Por ejemplo, si se implementa esta vitalidad como espera activa, cada objeto tiene una actividad interna consistente en la detección de si alguna de sus condiciones de disparo se satisface para activar el evento que corresponda, actuando así el propio objeto como agente.

   Como se indicaba en el esquema de clase, las relaciones de disparo tienen la estructura:

   destino::evento if condición-admisibilidad

   El mecanismo es similar al usado en Polka [38], con la diferencia fundamental de que en Polka todo son objetos (incluidos los mensajes), mientras que en O^3 los eventos (el equivalente al concepto de mensaje en Polka) no son objetos, sino propiedades de los objetos.

Otra propuesta que utiliza un concepto similar de disparo es la de OZ+ [135], un Sistema de Base de Datos OO donde se introduce el concepto de *regla auto-disparada*. Una regla auto-disparada es una regla sin parámetros que se ejecuta cuando todas las condiciones en ella definidas se satisfacen. Un monitor de objetos comprueba continuamente si las condiciones definidas en las reglas de los objetos se satisfacen para disparar, si ese es el caso, la regla en cuestión.

En Ode (un entorno de Base de Datos Orientada a Objetos) [49] los "triggers" también forman parte de la especificación de la clase y constan de dos partes: una condición y una acción. Están parametrizados, por lo que pueden ser activados muchas veces con diferentes valores. Las diferencias con las relaciones de disparo de Oasis son fundamentalmente tres:

(a) El entorno de especificación en Oasis es declarativo, mientras que en Ode tiene un marcado cariz imperativo. De hecho, una BDOO Ode es definida, consultada y manipulada por medio de un lenguaje de programación llamado O^{++}, que es una extensión del lenguaje de programación OO C^{++}

(b) En Oasis se pueden especificar distintas categorías de destinatario para el evento que se dispara

(c) Ode distingue explícitamente entre disparos contínuos (que una vez satisfecha su condición de disparo, continuan generando disparos mientras se siga cumpliendo tal condición) y disparos de 'una sola vez'. En Oasis tal distinción no existe, aunque se puede declarar un disparo de 'una sola vez' apoyándose en un atributo booleano adicional.

El evento a disparar (*evento* en la expresión anterior) puede ser cualquier evento del SO. Asimismo, para especificar quién es el destinatario, Oasis proporciona cuatro palabras reservadas:

(a) **self**: significa que el destinatario del evento disparado es el propio objeto que dispara.

(b) **object**: indica que el destinatario del evento disparado va a ser un objeto único, que debe ser identificado.[7]

(c) **class**. En este caso, el evento disparado va ser enviado a todas las instancias de la clase a la que pertenece el objeto 'disparador'.

(d) **society**: indica que el destino del evento que se dispara va a ser el conjunto de objetos de la Sociedad de Objetos que se está especificando.

Para cerrar este apartado, la especificación de las clases elementales *socio* y *libro* del SO Bibliotecario que se está utilizando como ejemplo, se presenta en sus versiones funcional, clausal y clausal con igualdad. Para incluir un caso de disparo, supóngase que cuando a un socio se le presta un libro y tiene otros 9 prestados, se le notifica que ya no puede sacar más libros. Es un ejemplo de disparo en el que el evento es el de notificación, el destinario es el mismo objeto que causa el disparo ('self') y la condición es que el libro que se le presta sea el décimo préstamo en activo sin que se le haya notificado ya tal situación.

Versión funcional: F-Oasis

```
conceptual_schema biblioteca

domains: nat,bool,time,string,set(X)

elementary class libro
   attributes
      constant
        codigo: nat key
        titulo: string
        fecha_entrada:time
        autor:set(string)
      variable
```

[7] Al desarrollar el entorno de prototipación, veremos que no será necesario pedir esa información si el evento disparado incluye en sus argumentos información suficiente sobre la identidad del objeto destinatario. En caso contrario, el Sistema la requerirá de forma automática.

```
        disponible(libro):bool (true)
        equations: b:libro;r:socio;v:bool;t,t1:time;
        disponible(b)=false if T=T1'prestar(b,r,t).
        disponible(b)=true if T=T1'devolver(b,r,t).
              else disponible(b) at T1.
    static constraints
        codigo<100.000
  events
    private
       adquirir(b,t) new.
       eliminar(b,t) destroy.
    shared
       prestar(b,r,t).
       devolver(b,r,t).
  preconditions
       T=T1'prestar(b,r,t) if disponible(b)=true.
       T=T1'devolver(b,r,t) if disponible(b)=false.
  triggering
end elementary class



elementary class socio
 attributes
   constant
     numero: string key
     nombre  : string
   variable
     num_libros(socio):nat (0)
       equations: r:socio;b:libro;n:nat;t:time;
        num_libros(r)=n if T=T1'prestar(b,r,t) and
                           num_libros(r)=n-1 at T1.
        num_libros(r)=n if T=T1'devolver(b,r,t) and
                           num_libros(r)=n+1 at T1.
                      else num_libros(r) at T1.
     avisado(socio):bool (false)
```

```
      equations: v:bool; t1:time;
        avisado(r)=true if T=T1'notificar(r,t).
                        else avisado(r) at T1.
   events
       private
          inscribir(r,t) new
          baja(r,t) destroy
          notificar(r,t)
      shared
          prestar(b,r,t)
          devolver(b,r,t)
   preconditions
          T=T1'baja(r,t) if num_libros(r)=0.
          T=T1'prestar(b,r,t) if num_libros(r)<10.
          T=T1'notificar(r,t) if num_libros(r)=10 and
                              avisado(r)=false.
   triggering
      self::notificar(r,t) if num_libros(r)=10.
end elementary class

end Conceptual Schema
```

Versión clausal: C-Oasis

```
conceptual_schema biblioteca

domains: nat,bool,time,string,set(X)

elementary class libro
   attributes
      constant
        codigo: nat key
        titulo: string
        fecha_entrada:time
        autor:set(string)
      variable
```

```
        disponible(libro,bool,time) (true)
        clauses: b:libro;r:socio;v:bool;t,t1:time.
        disponible(b,v,t) if  (prestar(b,r,t),v=false) or
                              (devolver(b,r,t),v=true) or
                              (t1 is t-1, disponible(b,v,t1)).
    static constraints
        codigo<100.000
  events
    private
       adquirir(b,t) new.
       eliminar(b,t) destroy.
    shared
       prestar(b,r,t).
       devolver(b,r,t).
  preconditions
        prestar(b,r,t) if disponible(b,true,t).
        devolver(b,r,t) if disponible(b,false,t).
  triggering
end elementary class


elementary class socio
  attributes
    constant
       numero: string key
       nombre  : string
    variable
       num_libros(socio,nat,time) (0)
       clauses: r:socio;b:libro;n:nat;t,t1:time;
       num_libros(r,n,t) if t1 is t-1,num_libros(r,n1,t1),
                            ((prestar(b,r,t),n is n1+1) or
                             (devolver(b,r,t),n is n1-1) or
                              n is n1).
       avisado(socio,bool,time) (false)
          clauses: v:bool; t1:time;
          avisado(r,v,t) if  (notificar(r,t),v=true) or
                              t1 is t-1,avisado(r,v,t1).
```

```
   events
       private
          inscribir(r,t) new
          baja(r,t) destroy
          notificar(r,t)
       shared
          prestar(b,r,t)
          devolver(b,r,t)
   preconditions
          baja(r,t) if num_libros(r,0,t).
          prestar(b,r,t) if num_libros(r,n,t),n<10.
          notificar(r,t) if num_libros(r,n,t),n=10,
                   avisado(r,false,t).
   triggering
      self::notificar(r,t) if num_libros(r,10,t).
end elementary class

end Conceptual Schema
```

Versión clausal con igualdad: L-Oasis

```
conceptual_schema biblioteca

domains: nat,bool,time,string,set(X)

elementary class libro
   attributes
      constant
        codigo: nat key
        titulo: string
        fecha_entrada:time
        autor:set(string)
      variable
          disponible(libro,time):bool (true)
          formulas: b:libro;r:socio;t:time.
          disponible(b,t)=false if prestar(b,r,t).
```

```
        disponible(b,t)=true  if devolver(b,r,t).
    static constraints
        codigo<100.000
  events
    private
       adquirir(b,t) new.
       eliminar(b,t) destroy.
    shared
       prestar(b,r,t).
     devolver(b,r,t).
  preconditions
       prestar(b,r,t) if disponible(b,t)=true.
       devolver(b,r,t) if disponible(b,t)=false.
  triggering
end elementary class


elementary class socio
   attributes
      constant
         numero: string key
         nombre: string
      variable
         num_libros(socio,time):nat (0)
            formulas: r:socio;b:libro;t:time;nombre:string;
            num_libros(r,t)=num_libros(r,t-1)+1 if prestar(b,r,t).
            num_libros(r,t)=num_libros(r,t-1)-1 if devolver(b,r,t).
         avisado(socio,time):bool (false)
            formulas:
            avisado(r,t)=true if notificar(r,t).
   events
       private
          inscribir(r,t) new
          baja(r,t) destroy
          notificar(r,t)
       shared
          prestar(b,r,t)
```

```
        devolver(b,r,t)
  preconditions
      baja(r,t) if num_libros(r,t)=0.
      prestar(b,r,t) if num_libros(r,t)<10.
      notificar(r,t) if num_libros(r,t)=10,
                avisado(r,t)=false.
  triggering
     self::notificar(r,t) if num_libros(r,t)=10.
end elementary class

end Conceptual Schema
```

3.1.3 Clases Complejas

A partir de dominios y clases elementales como componentes esenciales de la especificación de un SO, Oasis debe proporcionar mecanismos para describir objetos complejos.

Como herramienta de especificación en el contexto de O^3, Oasis da soporte sintáctico y semántico al uso de los cuatro operadores básicos de clases caracterizados en el capítulo anterior, como se muestra a continuación.

Un aspecto fundamental de toda clase compleja es que, independientemente del mecanismo seleccionado para construirla, se va a tratar finalmente de una clase más del Sistema de acuerdo con la definición de clase propuesta en el capítulo anterior.

Por esta razón, el esquema dado para una clase elemental es válido para cualquier clase que se especifica. Las clases complejas añaden a este esquema información sobre el operador usado como generador de la clase compleja. Cada operador hace que la nueva clase creada tenga ciertas ligaduras con respecto a las clases componentes, específicas de cada operador de acuerdo con su definición o que, opcionalmente, la nueva clase tenga determinadas propiedades emergentes (características del operador de clase utilizado).

Agregación

Una clase agregada en Oasis se define como:

complex class nombre-clase **aggregation of** {lista-clases}

De acuerdo con la definición de agregación en O^3, las principales características de la clase compleja agregada son:

1) Los eventos compartidos de las clases componentes de la agregación son eventos propios de la nueva clase. Estos eventos van a formar parte de la vida de los objetos componentes y del objeto compuesto por agregación.

2) De entre los eventos del punto anterior hay que señalar cual va a ser el evento creador de instancias de la agregación (*new*) y cual el que llevará los objetos a un estado de inexistencia (*destroy*).[8]

3) La clave de la agregación puede estar compuesta por las claves de las clases componentes o puede ser una propiedad emergente de la agregación. Es decisión del diseñador el decantarse por una u otra solución.

Un ejemplo claro de agregación lo tenemos en el SO de la Biblioteca para describir los préstamos. Un préstamo es una agregación de un libro (el prestado) a un socio (a quien se le presta). La nueva clase compleja tiene dos tipos distintos de propiedades:

* heredadas, como los identificadores de los componentes de la agregación y los eventos compartidos entre éstos.

[8]Es importante dejar claro que el efecto de un evento *destroy* no consiste en "borrar" el objeto. Por ejemplo, en el caso de la clase agregada *Préstamo*, cuyo evento *destroy* es *devolver*, el efecto de tal evento no es que

préstamo(l,s,t)=false

sino que se almacena ese evento como cualquier otro, para que tenga en cualquier instante sentido preguntarse por la existencia o inexistencia de un préstamo (información histórica) de acuerdo con fórmulas del estilo:

préstamo(l,s,t):- prestar(l,s,t1), t1<t, no(devolver(l,s,t2),t1<t2<t)

* emergentes como los nuevos atributos número del préstamo (elegido como clave de la agregación) y fecha del préstamo

La especificación en C-Oasis[9] se efectúa como se indica a continuación:

```
complex class prestamo aggregation of libro,socio
   attributes
      constant
         numero: string key
         fecha : string
      variable
   events
       private
          prestar(b,r,t) new
          devolver(b,r,t) destroy
   preconditions
...triggering
end complex class
```

Especialización/Generalización

Oasis proporciona los mecanismos de herencia básicos en todo contexto OO a través de los operadores de especialización y generalización definidos en el capítulo anterior.

Para caracterizar el tipo de herencia utilizado en las especificaciones Oasis, se va a partir del marco de referencia propuesto en [134], compuesto por cuatro dimensiones de diseño que permiten clasificar los mecanismos de herencia que proporciona cualquier lenguaje OO. Estos criterios son:

* **modificabilidad:** ¿De qué forma debe ser restringida la modificación de atributos heredados de una clase padre?

[9]No se da la especificación en las versiones funcional (F-Oasis) y clausal con igualdad (L-Oasis) para no cargar la presentación. Ya se ha establecido que la variación es mínima. En [37], [40] [60], [67], [78], [122], [96], [97], [109], [141] se presenta un amplio muestrario de especificaciones en las diferentes versiones de Oasis en distintos proyectos.

* **granularidad**: ¿Debe la herencia estar a nivel de clase o a nivel de objeto?

* **multiplicidad** ¿De qué forma se define y gestiona la herencia múltiple?

* **calidad** ¿Qué debe heredarse, código o comportamiento?

Estas dimensiones de diseño están muy orientadas a Lenguajes de Programación, como se aprecia claramente en el último criterio. Sin embargo, los tres primeros constituyen un conjunto de criterios válidos para clasificar mecanismos de herencia en Lenguajes de Especificación. De esta forma se puede determinar en qué punto del espacio tridimensional en el que se representa modificabilidad, granularidad y multiplicidad, se posiciona un lenguaje dado.

La primera dimensión de diseño, la **modificabilidad**, trata la herencia como un mecanismo de modificación incremental. Se distinguen cuatro mecanismos de modificación de atributos de una clase padre en la clase hijo que los hereda, que presentados de menor a mayor generalidad son los siguientes:

1) **compatibilidad de comportamiento**, consistente en que el comportamiento de la subclase es compatible con el de la superclase o clase padre. Se entiende como comportamiento compatible aquél en el que las instancias de la clase hija se comportan como instancias de la clase padre para todas las operaciones y argumentos definidos en ésta última.

2) **compatibilidad de signatura**, que requiere que la signatura de la clase hija sea sintácticamente compatible con la signatura de su clase padre. Este tipo de compatibilidad es usado en Lenguajes de Programación OO en forma de restricciones sobre las signaturas de las clases participantes comprobables en tiempo de compilación. La compatibilidad de signaturas se preserva con extensiones horizontales (adición de nuevas componentes a la signatura considerada) y verticales (restringiendo a un subtipo el tipo asociado a componentes de la signatura)

3) **compatibilidad de nombre**, en la que los nombres de operaciones de la clase padre se preservan en la clase hija, aunque puedan ser modificadas (especializadas).

4) **cancelación**, en la que se permite la modificación de atributos de la clase padre sin restricción alguna, pudiendo incluso eliminar algunos para tener clases hijas con menos atributos que sus clases padre correspondientes.

En el entorno de O^3, el comportamiento del operador de especialización garantiza la existencia de una compatibilidad de comportamiento, ya que la clase descendiente hereda eventos, atributos, precondiciones y relaciones de disparo de la clase padre, y las modificaciones que puede realizar son las siguientes:

* modificar la función de observación haciendo que un evento heredado tenga efectos adicionales sobre los atributos de la clase. Se tiene en este caso compatibilidad de comportamiento, porque el objeto hijo se comporta como un objeto padre si proyectamos su traza sobre el alfabeto de eventos del padre.

* modificar la precondición de algún evento del padre, añadiendo alguna nueva condición. Nuevamente se respeta la compatibilidad de comportamiento porque cualquier traza que representa objetos de la clase hija, puede ser proyectada sobre los eventos de la clase padre, dando como resultado una traza consistente de la clase padre.

La segunda dimensión de diseño se refería a la **granularidad** de la herencia. En Oasis es evidente que la herencia es a nivel de clase, dado que es un Lenguaje de Especificación de clases.

Por último, hay que fijar de qué forma se expresa en Oasis la herencia múltiple. Son dos los problemas fundamentales que hay que resolver en este contexto:

1) Cómo tratar el problema de la **herencia repetida** que surge cuando heredamos una misma propiedad desde dos padres distintos (una clase es antecesor de otra de más de una forma)

2) Solucionar la **colisión de nombres**, que surge cuando dos o más clases padre usan el mismo nombre para denotar propiedades distintas.

En el nivel de especificación en el que ahora estamos, estos problemas no se plantean. En el entorno de prototipación de Oasis, se soluciona el problema de la herencia repetida considerándola ilegal y forzando así al diseñador a desinstanciar la causa del error. Esta solución es la que adoptan muchos LPOO como Eiffel [84]. La colisión de nombres es detectada al compilar la especificación, siendo entonces posible renombrar el evento o el atributo que causa el problema.

El mecanismo sintáctico con el que se crea una clase especializada es muy simple:

complex class nombre-clase **especialization of** nombre-clase
[**where** condicion]

Al definir la especialización se distinguía entre especialización temporal y universal (dependiendo de que la clase especializada posea/no posea respectivamente su propio evento *new* y *destroy*), y entre dependencia e independencia de identificación (en función de que la clase especializada herede/no hereda respectivamente el identificador del padre). ¿Cómo se expresan estas situaciones en una especificación en Oasis?

La respuesta es simple. Si en la clase especializada se declara explícitamente un evento generador de instancias, etiquetado como siempre con la palabra reservada *new*, tenemos una especialización temporal. En caso contrario, tenemos una especialización universal, y deberá especificarse una **condición de especialización** que determine qué objetos pertenecen a la clase especializada considerada. Esta condición estará compuesta por expresiones del tipo atributo=valor.

Con respecto a la dependencia/independencia de identificación el planteamiento es inicialmente similar, pero referido a la clave. Si algún nuevo atributo de la clase especializada se etiqueta como clave se trata de una especialización con independencia de identificación. Si no es así, se tiene dependencia de identificación.

La especificación de una clase generalizada se realiza de la siguiente forma:

complex class nombre-clase **generalization-of** {lista-clases}

donde la lista de clases determina las clases base de la generalización.

La declaración de todas las posibles propiedades emergentes de la clase que se genera se hace a través del esquema general de clase, añadiendo en la sección que corresponda (events, attributes, preconditions, triggering) las nuevas propiedades de la clase compleja.

Para terminar este apartado, vamos a especificar la clase *Socio_no_fiable* que se comentó en el capítulo anterior,como una especialización de *Socio*, en la que sólo es posible sacar un libro simultáneamente de la Biblioteca por haber devuelto libros fuera de plazo en más de 10 ocasiones:

```
complex class socio_no_fiable especialization of socio
  attributes
     constant
     variables
  events
     private
       fichar(s,t) new
       perdonar(s,t) destroy
     shared
  preconditions
     prestar(l,s,t) if numero(s,t)=0
  triggering
end complex class
```

Como se desprende de esta especificación, se trata de una especialización con las siguientes características:

* dependencia de identificación, pues la clase especializada hereda la clave de la clase padre. Un socio no fiable tiene como identificador el que ya tenía como socio.

* especialización temporal: un socio adquiere el status de no fiable sólo cuando ocurre el correspondiente evento de creación *fichar*. Existirá como tal hasta que sea borrado explícitamente con una ocurrencia de *perdonar*

* Se modifica la precondición asociada al evento *prestar*, manteniendo compatibilidad de comportamiento con respecto al padre

porque si un evento *prestar* verifica la precondición definida en la clase especializada, verifica también la de la clase padre.

Asociación

El operador de asociación permite generar clases complejas cuyas instancias agrupan colecciones de instancias de la clase agrupada, a la que denotaremos como clase base.

Las relaciones entre entidades de aridad 1:M de la mayoría de los Modelos Semánticos se pueden diseñar en Oasis a través de este operador asociación.

Es importante no confundir el uso del operador asociación, con la definición de un dominio de tipo colección, genérico, que pueda ser asociado a atributos de una clase. Un ejemplo ayudará a distinguir claramente esta situación. Un departamento puede ser diseñado como una asociación de empleados. Es un uso típico del operador asociación como operador de clases. Sin embargo, el valor del atributo *autores* de la clase libro se diseña como un conjunto de cadenas de caracteres (set(string)), asumiendo que el TAD set(X) se define como dominio en la especificación. No es correcta la interpretación de que un libro es una clase compleja construida a partir de una hipotética clase autores, porque tal clase no existe. Autores es una propiedad (atributo) de la clase libro.

La declaración de una clase compleja asociada en Oasis se lleva a efecto de la siguiente forma:

complex class nombre-clase **association of** nombre-clase **using** tipo-colección [**grouped by** {lista-atributos}]

La palabra clave **using** permite seleccionar el mecanismo de agrupación que se desee de entre aquellos que inducen un orden entre sus elementos (lista, pila, cola) y los que no lo inducen (set, bag). La definición del atributo variable *miembros* que devuelve los elementos asociados de una instancia compleja en un instante dado, será la propia del mecanismo de agrupación que se elija. Esta definición es implícita, por lo que no es necesario declararla en la especificación. Dicha definición es de la forma:

```
members(i,c,t) if members(i,c',t-1),
                          ((insert(i,o,t), c=c'+{o}) or
                          (delete(i,o,t), c=c'-{o})).
```

siendo 'i' una instancia de la clase compleja asociada, 'o' un identificador de un objeto componente, '+'('-') es el mecanismo de inserción (borrado) propio del tipo de agrupación realizada, y 'c' los objetos asociados a la instancia i en un instante dado t.

Como se indicó al caracterizar en el capítulo anterior el operador asociación, si a la asociación se le da un criterio de agrupación en forma de una lista de atributos de la clase base, dicha lista de atributos pasa a ser un atributo constante de la clase compleja, que puede ser utilizado como atributo clave si el diseñador así lo decide.

Como ejemplo final, la declaración de una asociación de libros por un descriptor temático fundamental *microthesaurus* que supondremos declarado en la clase libro, con propiedades emergentes como la *sala* asignada a ese grupo de libros, sería:

```
complex class temas
 association of libro using set grouped by microthesaurus
     attributes
         constant
            microthesaurus:string key
         variables
            sala(temas,string,time) (null)
              clauses l:temas;x,s:string;t,t1:time;
               sala(l,x,t) if (asignar_sala(l,s,t),x=s) or
                              (t1 is t-1,sala(l,x,t1)).
     events
        private
            agrupar(l,t)  new
            desagrupar(l,t) destroy
            asignar_sala(l,x,t)
        shared
     preconditions
     triggering
end complex class
```

En este ejemplo se puede ver cómo entre las propiedades propias de la clase compleja *temas* están sus eventos propios de creación y destrucción de instancias, un evento de asignación de sala a un grupo creado y el correspondiente atributo variable asociado a él.

Como se hace uso de una condición de agrupación, la inserción de componentes es automática una vez creada la instancia compleja. Esto es consistente, pues al crear dicha instancia se le da valor a su clave, que es el atributo por el que se hace la asociación, con lo que ya se sabe qué instancias de la clase base estarán en cada una de las instancias complejas (es decir, qué libros estarán en cada objeto de la clase asociada *temas*).

Composición Paralela

La composición paralela de dos clases se declara en Oasis como:

complex class nombre-clase **parallel composition of** {lista-clases}

con las propiedades vistas en el capítulo anterior.

El operador de composición paralela va a resultar especialmente útil para definir un SO completo como la composición paralela de las clases componentes. Así, de forma implícita, una especificación de un SO es de hecho una instancia de la composición paralela de todas las clases participantes, lo que sintácticamente se representa de forma simplificada con la sintaxis habitual:

```
Conceptual Schema
domains
   ...
elementary classes
   ...
complex classes
   ...
End Conceptual Schema
```

en lugar de escribir formalmente:

Complex Class Conceptual Schema *parallel composition of*
({domains},{elementary classes},{complex classes})

De esta forma podemos dar una definición constructiva e inductiva de Esquemas Conceptuales (EC):

Definición 3.1 *Se define inductivamente un Esquema Conceptual como:*

1) Una clase elemental es un EC.

*2) Dado un conjunto OP de operadores entre clases (*aggregation, especialization, generalization, association, parallel composition*), y dados dos EC A y B, entonces A OP B es un EC*

3) Sólo se pueden obtener EC a partir de las reglas anteriores.□

Se da así soporte a un estilo de especificación incremental con el que el grado de reusabilidad de clases definidas previamente es muy alto. El proceso de composición de clases culmina cuando se llega a la clase compleja que representa el SO como Sociedad de Objetos Interactivos.

3.2 Otras propuestas de Lenguajes de Especificación

Vamos a comparar en esta sección Oasis con otros Lenguajes de Especificación diseñados para describir SI bajo un modelo OO.

Los lenguajes de especificación seleccionados para la comparación en base a la proximidad de sus objetivos y del contexto de especificación son los siguientes:

1) **Troll**, [70,72,102,107,69], desarrollado dentro del entorno del grupo IS-CORE como evolución con sintaxis textual de Oblog.

2) **MAL-2** ([100,101]), consecuencia del trabajo del grupo 'New Forest' (Imperial College (London)).

3) **CMSL** [138,136,137], un lenguaje de especificación que combina álgebra de procesos [10,11,12] con especificaciones algebraicas de tipos abstractos de datos para especificar objetos abstractos dinámicos.

Troll

Empezamos con Troll [70], un Lenguaje para especificación OO de SI, desarrollado por G.Saake y C.Sernadas entre otros, que es continuación de las ideas introducidas por OBLOG ([72,117]) y que constituye actualmente la versión textual [10] más moderna de dicho lenguaje. El cuerpo de la definición de un objeto lo constituye su maqueta '**template**'. Una maqueta de objeto puede incluir los siguientes apartados:

template [nombre maqueta]
- **data types** signaturas de tipos de datos importados
- **attributes** nombres de atributo y declaraciones de tipo
- **events** nombres de evento y declaraciones de parámetros
- **constraints** restricciones estáticas y dinámicas sobre valores de atributos
- **derivation** reglas para derivar atributos y eventos
- **valuation** efectos de eventos sobre atributos
- **behavior**
  - **permissions** precondiciones para la ocurrencia de eventos
  - **obligations** requerimientos de completitud para los ciclos de vida [11]
  - **commitments** metas a corto plazo dependientes del estado [12]
  - **patterns** transacciones y 'scripts' [13]

end template [nombre maqueta]

Un objeto individual se define mediante un nombre propio y una maqueta. Una clase de objetos se define como un nombre de clase, una maqueta y un mecanismo de identificación. En Troll se declaran **identificadores externos**. Estos identificadores externos son elementos del soporte de un TAD. Pueden ser tuplas de valores atómicos de

[10] Los últimos trabajos en OBLOG tienen como objetivo el desarrollo de un lenguaje gráfico (visual) que posea una capacidad expresiva similar a la de Troll.

[11] Estos requerimientos de completitud deben satisfacerse para que el objeto pueda morir.

[12] Este apartado describe actividad interna de los objetos, declarando una especie de relaciones de disparo a través del operador temporal 'after'. Por ejemplo, after(evento [and condición]) then evento.

[13] En este apartado se definen procesos utilizando un lenguaje de procesos tipo CSP.

datos. El conjunto de identificadores externos y maqueta constituyen el tipo de la clase. Dado un identificador externo y un nombre de clase, se tiene definido un mecanismo de identificación único para instancias de esa clase. El conjunto de identificadores de una clase constituye su **espacio de identificación**.

Los mecanismos de estructuración de clases suministrados por Troll son **'roles', especialización, generalización** y **agregación.**

* El concepto de 'role' denota una especialización que temporalmente hace que se tenga una vista especial del objeto (que ha empezado a desempeñar el 'role' de...)

* La especialización en Troll es un caso particular de 'role' en la que el objeto especializado desempeña ese 'role' desde el instante en que es creado. Se trata de una relación 'Is-a' activa desde el momento en que se crea el objeto padre.

* La generalización es un mecanismo similar a la especialización, pero que funciona en sentido contrario.

* La agregación permite construir objetos a partir de componentes (objetos complejos). Troll distingue dos tipos de objetos complejos:

  1) *objetos complejos disjuntos* cuando los componentes no pueden existir fuera del objeto complejo.
  2) Los *objetos complejos no disjuntos* pueden compartir componentes. Pueden ser a su vez de dos tipos:
     (a) *dinámicos*, si su composición cambia en el tiempo como consecuencia de la ocurrencia de eventos.
     (b) *estáticos*, cuya composición se describe a través de predicados estáticos.

  En cualquier tipo de objeto complejo, las componentes están encapsuladas en el sentido de que su estado solo puede ser alterado por eventos locales a dichas componentes. Sus atributos, sin embargo, son visibles. La coordinación y sincronización entre el objeto complejo y sus componentes debe realizarse a través de una comunicación.

Para describir Sistemas de Objetos Interactivos los mecanismos de estructuración anteriores no son suficientes. Troll introduce:

* **Relaciones**: describen cómo se conectan las diversas componentes del Sistema, a través de la declaración de:

    1) *interacciones globales*, que describen comunicación entre objetos (sincronización de eventos)

    2) restricciones globales, en aquellos casos en que una restricción no se puede hacer local a ningún objeto.

* **Interfaces**, que describen mecanismos de control de acceso a objetos.

Podríamos dedicar todo un capítulo a presentar Troll, pero no ese el objetivo de este apartado. Vistas sucintamente sus características fundamentales, vamos a compararlo críticamente con Oasis. Tal comparación es sin lugar a dudas muy interesante. Oasis y Troll se encuentran en el 2nd International Workshop of the IS-CORE group, en el Imperial College (Londres,Septiembre-1991), poniéndose de manifiesto que se trata de dos aproximaciones con objetivos similares, con muchas ideas en común en cuanto a su capacidad expresiva, pero también con diferencias cualitativas importantes.

Ambas propuestas presentan un lenguaje de especificación OO, basado en la Lógica para describir propiedades y comportamiento de sociedades de objetos interactivos.

Las diferencias esenciales son las siguientes:

* Siguiendo el criterio de clasificación propuesto por Olivé y Bubenko en [22], Oasis adopta una aproximación deductiva, mientras que Troll tiene una apariencia más dinámica. Las implementaciones de Oasis generan entornos deductivos, mientras que Troll parece propiciar la creación de Sistemas Dinámicos.

* Las especificaciones en Troll no son ejecutables (por el momento), frente a la ejecutabilidad de las especificaciones Oasis, propiedad fundamental dentro del entorno de producción automática de Software del que forman parte.

* El modelo OO subyacente en Troll utiliza como soporte formal matemático la Teoría de Categorías [42,41,44], mientras que en Oasis el modelo OO subyacente (O^3) fusiona conceptos básicos de SI con definiciones ontológicas dentro de un marco algebraico.

En cuanto a su capacidad expresiva, son muchos los conceptos compartidos:

* Troll permite definir objetos y clases indistintamente. En O^3 los objetos se agrupan en clases a través de un proceso de clasificación, luego Oasis es un lenguaje de definición de clases. No obstante, un objeto siempre puede definirse como instancia de una clase cuyo tipo lo compone sólo dicho objeto.

* Ambos utilizan atributos y eventos para dar cuenta de aspectos estructurales (de estado) y de comportamiento respectivamente

  Sin embargo, una diferencia importante se deriva de la definición en Oasis de eventos compartidos. Su falta fuerza en Troll la introducción de relaciones de interacción con las que sincronizar eventos de diferentes clases.

* Los *data types* de Troll son las *clases primitivas* de Oasis. En Troll hay que declararlos en cada clase (objeto) mientras que en Oasis sólo se declaran una vez, al principio de la especificación del Sistema.

* El mecanismo de identificación de objetos de Troll se simplifica en Oasis con el uso de la palabra reservada **key** en los atributos constantes componentes de la clave.

* Ambos permiten especificar restricciones de integridad estáticas asociadas a valores de atributos. Troll introduce una lógica temporal para definir también restricciones de integridad dinámicas que involucran tanto a atributos (*constraints*) como a eventos (*obligations*). Estas restricciones establecen relaciones inter-estado. En Oasis también se han introducido tales restricciones en algunas versiones, y se está desarrollando un intenso trabajo para disponer de un entorno en el que se incluyan fórmulas temporales

como una componente más del lenguaje de primer orden de la Teoría equivalente.

* Troll introduce una sección especial para definir atributos derivados. En Oasis la definición de tales atributos no está implementada, aunque inminentes extensiones del lenguaje la incluirán también en la forma de una nueva sección.

  Además, en Troll todos los atributos y eventos de una clase generalizada se derivan de los atributos y eventos correspondientes de las clases bases, lo que permite crear sinónimos. Oasis fuerza en ese caso a dar igual nombre y aridad a los eventos comunes.

* La sección de evaluación de Troll introduce los efectos de un evento sobre los atributos afectados, haciendo uso de un enfoque dinámico. Dado un evento, deben conocerse todos los atributos que serán modificados por una ocurrencia del mismo. En Oasis, la definición de los atributos es deductiva, debiendo explicitarse en la definición axiomática de un atributo, todos los eventos que puedan modificar su valor.

* El comportamiento de los objetos lo caracteriza en Troll las precondiciones asociadas a eventos (*permissions*), unas relaciones de disparo simplificadas (*commitments*) y los *patterns* que definen procesos utilizando un álgebra de procesos. En Oasis el comportamiento se caracteriza a través de precondiciones y relaciones de disparo.

  Las precondiciones en Oasis son similares a los *permissions*, aunque en Troll la potencia expresiva es mayor al introducir una lógica temporal. Las relaciones de disparo, sin embargo, son más expresivas que los *commitments* de Troll porque:

  1) un evento puede ser activado por la satisfacción de una condición (no sólo por otro evento como en Troll)
  2) la granularidad del destinatario del evento disparado en Oasis ofrece al diseñador la posibilidad de elegir entre enviar el evento al mismo objeto que los dispara (*self*), a otro objeto

del Sistema (*object*), a todos los objetos de una clase (*class*) o a todos los objetos de la Sociedad de Objetos (*society*).

* En Oasis la introducción de operadores entre clases da cuenta de una forma simple y potente de la estructura de clases complejas. Proporciona asimismo un mecanismo constructivo para definir una Especificación de un SO como la clase compleja generada por composición paralela de todas las clases componentes. En Troll se introducen diferentes tipos de mecanismos (de estructura y de relación) para dar cuenta de la Sociedad de Objetos en su conjunto.

  En Troll, la especialización de Oasis se divide en 'role' y especialización. Lo que en Troll se llama 'role' en Oasis es un caso particular de especialización (temporal) fácilmente detectable por la existencia de un evento de creación propio en la clase especializada.

  La Agregación en Oasis va asociada a un evento compartido que la caracteriza. En Troll se introducen relaciones globales entre objetos (interacciones y restricciones) adicionales para dar cuenta de la misma noción.

  El operador de Composición Paralela en Oasis permite siempre asociar restricciones localmente a una clase, por lo que no es necesario introducir restricciones globales como en Troll.

  Oasis proporciona además un operador de asociación que no posee Troll. Troll introduce para ello la noción de agregación dinámica, en la que se puede definir atributos evaluados como colección (set, etc.), debiendo el diseñador especificar explícitamente los eventos de inserción y borrado de nuevos componentes en el valor de ese atributo, en el contexto de los llamados en Troll *objetos complejos no disjuntos.*

* El uso de operadores de clase en Oasis no hace necesario distinguir entre objetos complejos disjuntos y no disjuntos, o entre agregación estática y dinámica. El uso del operador adecuado da cuenta de esas nociones de Troll de una forma homogénea, manteniendo un estilo incremental y constructivo en la definición de

clases complejas.

* Troll permite definir procesos a partir de eventos utilizando un álgebra de procesos. Proporciona así un mecanismo adicional para describir el comportamiento de los objetos. La noción de proceso en Oasis depende del nivel de abstracción. Como la definición de atributos se hace de forma declarativa, la agrupación de eventos atómicos en un proceso puede definirse como un nuevo evento, dando cuenta de su efecto global sobre los atributos implicados en sus correspondientes definiciones. No obstante, la introducción del concepto de proceso y de un álgebra de procesos asociada enriquecería la capacidad expresiva del lenguaje.

* La declaración de interfaces en Troll es un mecanismo para describir control de acceso a objetos, proyectando símbolos de eventos y atributos sobre símbolos visibles externamente. Permiten asimismo especificar distintas vistas sobre los objetos restringiendo el tipo de una clase a algún subconjunto seleccionado. En Oasis, no se declaran explícitamente interfaces. El sustrato ontológico de O^3 establece que la interface de todo objeto de una clase está compuesta por sus eventos y atributos. Lo que no se contempla en el estado actual del lenguaje es la asignación de *agentes* a los eventos en el caso más general[14]. Próximas extensiones del Lenguaje van en esa dirección.

MAL-2

MAL-2 ([101]) es una extensión OO de MAL ([100]), realizada por los miembros del equipo *New Forest.*

En MAL-2 cada especificación de objeto consta de una colección de acciones (*actions*) y atributos (*attributes*) que constituyen su *signatura.* Además, el lenguaje de especificación determina qué acciones y atributos son locales, y cuales son compartidas. Si una acción se declara local a un objeto, se genera un axioma que establece que la acción sólo puede modificar atributos del objeto. Análogamente, un atributo local tiene

[14]Los disparos constituyen una excepción, ya que el agente asociado a un disparo es el objeto que lo activa por satisfacción de la condición de disparo correspondiente

asociado un axioma que indica que sólo puede ser modificado por acciones pertenecientes al objeto definido. A estos axiomas se les llama genéricamente *axiomas de localidad.*

Estos axiomas de localidad no forman parte explícita de la especificación. Forman parte de las *presentaciones de teoría* denotadas por la especificación. Se tiene un sistema con tres niveles que son:

1) textos de especificación: son textos del lenguaje de especificación, que describen el comportamiento de una colección de objetos interactivos.

2) presentaciones de teoría: el texto anterior denota una familia de presentaciones de teorías (una presentación por cada objeto especificado), conectadas a través de morfismos.

3) objetos: un modelo de una presentación de teoría es un objeto que satisface la presentación.

Vamos a centrarnos en el lenguaje de especificación, que es lo que nos interesa en este apartado.

La especificación de un objeto tiene la siguiente estructura:

object name
uses type,type,...;
l-attributes
símbolo-atributo:$sort \times \ldots \times sort \rightarrow sort$;
$\vdots$
l-actions
símbolo-acción:$sort \times \ldots \times sort$;
$\vdots$
s-attributes
símbolo-atributo:$sort \times \ldots \times sort \rightarrow sort$;
$\vdots$
s-actions
símbolo-acción:$sort \times \ldots \times sort$;
$\vdots$
importing subobject **via** nombre-morfismo **where**

```
        acción-subobjeto is acción-objeto;
        ⋮
        acción-subobjeto is acción-objeto;
        ⋮
⋮
axioms
        axiom;
        ⋮
end
```

donde los axiomas expresan de forma dinámica el efecto de un evento sobre los atributos afectados, y los aspectos estructurales de la Sociedad de Objetos se representan mediante una relación *subobjeto* caracterizada a través de la noción de morfismo entre especificaciones de objeto ([101]). En pocas palabras, X es un subobjeto de Y si existe un morfismo de la especificación de X a la especificación de Y. Ello implica que todas las acciones y atributos de X son (a través de un cambio de nombre) acciones y atributos de Y. Así se caracterizan mecanismos estructurales como herencia y agregación:

* La herencia se representa mediante un único morfismo. Si Y hereda X, entonces existe un morfismo de la especificación de X a la especificación de Y.

* La agregación se captura con la ayuda de dos morfismos. Si X+Y es la agregación de X e Y, entonces existen morfismos desde X a X+Y y desde Y a X+Y. La agregación puede llevar asociada la identificación de acciones y atributos entre X e Y, en cuyo caso se crea una abstracción Z de X+Y.

De la comparación con Oasis se deducen aspectos interesantes. Como en el caso de Troll, MAL-2 toma una aproximación dinámica, frente a la declarativa de Oasis, y no es ejecutable, lo que marca de nuevo una diferencia cualitativa importante.

Los tipos declarados en MAL-2 al principio de la especificación (en el apartado *uses*) se corresponden con los dominios o clases primitivas de Oasis.

MAL-2 es un lenguaje de especificación de objetos, no de clases. No deja de ser curioso que para especificar un Sistema compuesto por n objetos de una misma clase haya que elaborar n especificaciones de objetos.

Además, MAL-2 utiliza un fundamento formal categórico, incluyendo la noción de morfismo explícitamente en la especificación. Esto no es ni mucho menos intrínsecamente negativo, pues permite disponer de un soporte formal muy potente, pero dada la complejidad asociada a la Teoría de Categorías, su introducción en el lenguaje de especificación no parece ser lo más conveniente. La estructura de la Sociedad de Objetos se especifica a través de esa noción de morfismo, frente al mecanismo constructivo algebraico basado en el uso de operadores de clase proporcionado por Oasis.

En lo que se refiere a la signatura de los objetos, las acciones y atributos de MAL-2 se corresponden con los eventos y atributos respectivamente de Oasis. MAL-2 introduce atributos compartidos, que son aquéllos que pueden ser modificados por eventos de otro objeto. Estos eventos, a su vez, se declaran compartidos por esa razón. En Oasis, un evento compartido pertenece a la signatura de las clases que lo comparten, por lo que este problema se soluciona de forma elemental.

Por último, los axiomas de localidad denotados por las especificaciones MAL-2 no son necesarios en Oasis, pues son intrínsecos al modelo OO utilizado. Los atributos de una clase sólo son modificados por los eventos declarados en dicha clase.

CMSL

CMSL es un lenguaje de especificicación de esquemas conceptuales que combina álgebra de procesos y especificaciones algebraicas de TAD's para especificar sociedades de objetos dinámicos abstractos.

Como en Oasis, el lenguaje se construye sobre un marco conceptual OO bien definido, presentado en [138]. Sus principios fundamentales son, muy brevemente, los siguientes:

1) Distingue entre *objetos* (algo que puede ser colocado en un conjunto y que es contable) y *masas* (no contable, representable a través de medidas). En un Modelo Conceptual (MC) sólo se representan objetos.

2) Un MC es estrictamente OO si todo objeto tiene un único identificador global.

3) Una *clase natural* es el mayor conjunto de instancias que satisfacen un conjunto de generalizaciones empíricas. La clase es la extensión de un tipo. Una clase puede ser:

   (a) *contingente*, si un objeto que es una instancia de tal clase, puede no serlo en otros estados del Universo de Discurso. A las clases contingentes se les llama 'roles'

   (b) *esencial*, cuando no es contingente.

4) Los aspectos estructurales de un objeto los determinan sus *atributos*. La estructura de un objeto puede ser también esencial o contingente.

5) Los cambios de estado se modelan a través de la noción de *evento*. Cada evento es una función de los valores de los atributos, y no tiene estados intermedios.

6) La comunicación entre objetos se efectúa introduciendo:

   * *encapsulación*[15]: los eventos encapsulados reciben el nombre de *mensajes*
   * una *función de comunicación*: si un evento e1 debe comunicarse con un evento e2, entonces

     e1 | e2=e3

     significa que e3 es un evento que consiste en la ejecución síncrona de e1 y e2.

7) Por último, un proceso se define para cada clase natural haciendo uso de un álgebra de procesos tipo CSP.

[15]Encapsulación significa aquí proteger del entorno: un evento encapsulado sólo puede ocurrir como parte de una comunicación. Esta noción de encapsulación es distinta de la utilizada habitualmente en entornos OO, que se refiere a localidad de estados y eventos.

* La actividad interna propiciada por las relaciones de disparo de Oasis no tiene equivalente en CMSL.

Presentado el modelo objetual O^3 y el Lenguaje de Especificación Oasis como herramienta descriptiva asociada a dicho modelo, y efectuada la comparación con otras propuestas relevantes, vamos a ver en el capítulo siguiente cómo formalizar el entorno de especificación caracterizado por el Lenguaje Oasis. Posteriormente veremos cómo desarrollar un entorno de programación automática demostrando que las especificaciones de SO escritas en Oasis tienen una propiedad lógico-formal fundamental: se corresponden con teorías de primer orden (clausales en el caso de C-Oasis, ecuacionales en el caso de F-Oasis y clausales con igualdad en el caso de L-Oasis). Ello sentará las bases del desarrollo de entornos de prototipación basados realmente en métodos formales.

Capítulo 4

Ejecutabilidad de una especificación Oasis en un entorno lógico

4.1 Introducción

Vamos a ver en este capítulo que una especificación en Oasis es equivalente a una Teoría de Primer Orden, clausal para la versión clausal de Oasis (C-Oasis), ecuacional para su versión funcional (F-Oasis) y clausal con igualdad para su versión clausal con igualdad (L-Oasis). De esta forma se dispone de un entorno de trabajo formal: asociada a una especificación en Oasis tendremos una Teoría cuyo modelo es la extensión de una Base de Datos OO extendida (semántica declarativa) y en la que el mecanismo deductivo utilizado determinará la semántica operacional.

Esa BDOO se implementa sobre:

1) el modelo estándar de la teoría clausal a la que es equivalente la especificación en C-Oasis.

   La semántica operacional se implementa a través del mecanismo de resolución SLDNF.

2) el modelo inicial de la teoría ecuacional a la que es equivalente la especificación en F-Oasis.

   La semántica operacional en este caso se implementa a través de mecanismos de reescritura de términos o de 'narrowing' condicional.

3) el modelo inicial de la teoría clausal con igualdad a la que es equivalente la especificación en L-Oasis.

   La semántica operacional se implementa a través de flat-SLD-resolución.

Un entorno tal va a permitir generar de forma automática un Prototipo del Sistema que será equivalente a la Especificación. Este Prototipo es el Programa Lógico (clausal, ecuacional o clausal con igualdad) usado para representar la Teoría de Primer Orden correspondiente.

Las Teorías de Primer Orden clausales y ecuacionales asociadas a una especificación en Oasis se presentan en las próximas secciones.

Además, si una especificación de un SO en Oasis es equivalente a una Teoría de Primer Orden, podemos usar su Lenguaje de Primer

Orden asociado como lenguaje de usuario de esa BDOO. Se dispone así de un potente lenguaje de consulta en un entorno formal.

La evolución del Sistema se va a traducir en la adición a la Teoría correspondiente de los nuevos eventos relevantes que ocurran. Caracterizaremos tales aspectos dinámicos a través de dos aproximaciones:

* utilizando metaprogramación en un entorno lógico clausal. Una especificación se representará mediante un Metaprograma Lógico, y un metapredicado de inserción dará cuenta de la inserción de nuevos eventos en la traza del Sistema.

* a través de operadores algebraicos (eventos) que cambian el estado de los objetos, desde una aproximación algebraica.

4.2 Entorno clausal:C-Oasis

Esta sección presenta la sintaxis y la semántica asociada a las fórmulas bien formadas (fbf) de una Teoría de Primer Orden Clausal equivalente a una especificación en C-Oasis. Definiciones y nociones básicas del marco lógico utilizado pueden encontrarse en [77].

El entorno es clausal normal: hace uso de *cláusulas normales*, que son aquellas cláusulas de la forma

$$A \leftarrow B_1, \ldots, B_n$$

con un único átomo en su cabeza y n literales (átomos positivos o negativos) en su cuerpo.

La lógica de primer orden tiene dos aspectos: sintaxis y semántica. El aspecto sintáctico se preocupa de caracterizar fbf's generadas por la gramática de un lenguaje formal, así como por los mecanismos sintácticos de demostración asociados. La semántica hace referencia a los significados asociados a las fbf's y a los símbolos que contiene.

Definición 4.1 *Una* **Teoría de primer orden** *consiste en un alfabeto, un lenguaje de primer orden, un conjunto de axiomas y un conjunto de reglas de inferencia ([83] [118]). El lenguaje de primer orden lo conforman las fbf's de la Teoría. Los axiomas son un conjunto determinado de fbf's. Axiomas y reglas de inferencia se utilizan para deducir los teoremas de la Teoría.*

Además de los componentes anteriores, si la Teoría de Primer Orden es tipada, tenemos una componente adicional que es un conjunto finito de tipos. De acuerdo con las definiciones básicas del entorno O^3, hablaremos de un conjunto finito de clases (provenientes de la especificación fuente Oasis), y de Teoría de primer orden tipada (o clasificada en nuestro caso).

4.2.1 Teoría Clausal de Primer Orden asociada a una especificación en C-Oasis

Vamos a ver en primer lugar el **Lenguaje de Primer Orden** de la Teoría Clausal equivalente a una especificación en C-Oasis.

El lenguaje L de primer orden de la Teoría Clausal de Primer orden clasificada asociada a una especificación en C-Oasis se define inductivamente de la siguiente forma, a partir de un alfabeto compuesto por variables, constantes, símbolos de predicado y cuantificadores (todos ellos clasificados) proporcionados por la especificación fuente en Oasis:

Definición 4.2 *Los* **términos** *de clase d se definen inductivamente como:*

1) Una constante de clase d es un término de clase d

2) Una variable de clase d es un término de clase d

3) Solo los términos anteriores son de clase d

Definamos ahora los **átomos** del lenguaje de primer orden:

Definición 4.3 *Los átomos del lenguaje L se definen inductivamente como:*

1) Si C es una clase con atributos constantes $a_1, ..., a_n$ de clases $C_1, ..., C_n$, y $t_1, ..., t_n$ son términos de clase $C_1, ..., C_n$ respectivamente, entonces $C(t_1, ..., t_n, t)$ es un átomo.

Estos átomos representan instancias de objetos existentes. Por ejemplo, dada la clase libro anterior con atributos constantes código, título de clase string, **libro('b1','Hamlet',t)** *es un átomo.*

2) *Dado el conjunto ops=*$\{<, >, \leq, \geq, is\}$ *de operadores relacionales. Si f,g son términos de la misma clase (primitiva) y para dicha clase los operadores ops están definidos, entonces* f **op** g *es un átomo si* **op** $\in$ *ops .*

Estos átomos permiten efectuar sencillas operaciones relacionales entre atributos y/o valores de atributos. Por ejemplo, **n<10**.

3) *Si C es una clase, a es un atributo de C de clase A, x es un término de clase A,* k_c *es la clase asociada al atributo clave de C y* i_{k_c} *es una de sus instancias, entonces* $a(i_{k_c}, x, t)$ *es un átomo.*

Estos átomos se utilizan para representar valores de atributos.

Por ejemplo, sea disponible *un atributo de la clase libro de clase boolean. Si 'b1' es una instancia de la clase libro, entonces*

disponible('b1',v,t)

es un átomo.

4) *Si e es un evento de aridad n+1, con argumentos de clase* $C_i \mid i : 1 \ldots n$ *y* $x_1, ..., x_n$ *son términos de clase* $C_1, ..., C_n$ *entonces* $e(x_1, ..., x_n, t)$ *es un átomo.*

Por ejemplo, **prestar('b1','l1',t)**

5) *Sólo son átomos de L los obtenidos a través de las reglas anteriores.*

Sólo falta ahora determinar las fbf's de L.

Definición 4.4 *Las fbf's de L se definen inductivamente de la siguiente manera:*

1) *Un átomo de L es una fbf.*

2) *Si F,G son fbf's, entonces not(F), F and G, F or G,* $F \leftarrow G$ *son fbf's.*

3) *Sólo son fbf's de L las obtenidas a partir de las dos reglas anteriores.*

Definido el lenguaje de primer orden de la Teoría equivalente a la especificación, vamos a ver cuales van a ser sus axiomas:

Definición 4.5 *La Teoría clausal equivalente a una especificación en Oasis va a contener siete tipos de axiomas:*

* **Axiomas referentes a clases primitivas (dominios).** *Estos axiomas incorporan a la Teoría los tipos primitivos: nat(0), nat(1), ..., bool(true), ..., de forma extensional o intensional. En el entorno de prototipación implementado, los más usuales están predefinidos. Proporcionan, como vimos al presentar Oasis, el soporte básico a partir del cual se desarrolla el conjunto de clases elementales y complejas del Sistema, ya que se usan como clases identificadoras de objetos y como clases asociadas a atributos.*

* **Axiomas de clase** *Para cada clase elemental o compleja del Sistema existe un axioma de clase construido como se indica a continuación.*

  Sea k_c el conjunto de atributos constantes de una clase C, y sean $new_c, destroy_c$ los eventos creadores y destructores de instancias de la clase C. Entonces tenemos el axioma de clase:

$$C(k_c,t) \leftarrow new_c(C(k_c,t1)), t1 < t, not(destroy_c(C(k_c,t2)), t2 < t, t1 < t2)$$

* **Axiomas de restricciones estáticas de integridad.** *Son fbf's cerradas de L constituidas por conjunciones de literales construidos a partir de atributos, u operaciones relacionales cuyos operandos son términos de la clase de un atributo (es decir, átomos de los tipos 2,3 respectivamente).*

  Por ejemplo, **número(r,n,t) and n<10**

* **Axiomas de eventos relevantes.** *Si $e(c_1,...,c_n,c_t)$ es un evento relevante, y $c_1,...,c_n,c_t$ son constantes de las clases de sus argumentos, entonces $e(c_1,...,c_n,c_t)$ es un axioma de la Teoría.*

  Por ejemplo, **prestar('b1','l1',t)**

* **Axiomas de precondiciones de eventos.** *Estos axiomas se introducen como restricciones de integridad. Son fbf's de L de la forma $A \leftarrow B$ que verifican que A es un átomo de tipo 4 (un evento). Es decir, son cláusulas en cuya cabeza aparece un evento, y cuyo cuerpo es una conjunción/disyunción de literales.*

  Por ejemplo,

  prestar(b,r,t) if disponible(b,true,t) and número(r,n,t) and n<10.

* **Axiomas de definición de atributos.** *Por cada objeto de una clase se añaden los siguientes axiomas:*

  1) Un axioma por atributo constante: dado el atributo constante de la clase C a_c, de clase X, y siendo x una constante de clase X, tenemos $a_c(C,x,t)$ como axioma de evaluación del atributo constante.[1]

  Un ejemplo de un axioma de este tipo es

  título('b1','Hamlet',6)

  2) las cláusulas que conforman la definición de los atributos variables. Estas cláusulas tienen como cabeza el atributo variable que se define, y como cuerpo la fbf que constituye su definición.

  Por ejemplo, un axioma de la teoría para el ejemplo de la Biblioteca es la definición del atributo variable número *de libros de la clase* socio*:*

```
numero(r,n,t) if t1 is t-1,numero(r,n',t1) and
        ((prestar(b,r,t) and n is n'+1) or
        (devolver(b,r,t) and n is n'-1))
```

* **Axiomas de relaciones de disparo.** *Son cláusulas que tienen en cabeza un átomo construido a partir de un evento. Tienen por*

[1] Estos atributos son redundantes porque esa información viene incluida en el evento generador de instancias de la clase. No obstante, se añaden para disponer de un tratamiento común para los atributos.

tanto la misma estructura de las precondiciones, pero como veremos no tienen que ver con la teoría sino con sus modificaciones

evento[2]si condición

donde condición *es una fbf de L compuesta por conjunción y disyunción de literales.*

Axiomas de precondiciones y de disparo van a jugar un papel fundamental al hablar en términos de animación de la Teoría resultante de la especificación.

Si tomamos como reglas de inferencia de la Teoría los mecanismos de resolución implícitos en cualquier intérprete Prolog, tenemos la Teoría clausal buscada, correspondiente formalmente a la especificación en Oasis a partir de la cual se genera mediante un proceso de traducción.

Esta Teoría tiene como modelo una Base de Datos OO que va a ser utilizada para evaluar las fórmulas bien formadas (fbf's) del lenguaje de usuario que se presenta en el próximo apartado.

La generación automática de la Teoría y su representación en Prolog a partir de una especificación Oasis se explica con detalle en [141,25]. Estos traductores de alto nivel, junto con sus editores gráficos asociados y el correspondiente entorno dinámico de animación constituyen una implementación operativa del entorno de producción automática de Software presentado en esta Tesis.

4.2.2 Lenguaje de Primer Orden asociado a una especificación en C-Oasis como lenguaje de usuario

Sintaxis

El lenguaje L' que se pone a disposición del usuario para que sea utilizado como lenguaje de consulta es un subconjunto de L, que incluye:

[2]La referencia al destinatario del evento que se dispara va a ser externa a la Teoría que se está construyendo. Esa información la tendrá el animador para saber quienes serán los receptores de un evento activado por el propio Sistema.

* Los términos de L' son los mismos que los de L

* Los átomos de L' son los definidos para L.

* Las fbf's de L' se definen inductivamente de la siguiente forma:

  1) Un átomo de L' es una fbf.
  2) Si F,G son fbf's de L', entonces not(F), F and G, F or G son fbf's
  3) Solo las obtenidas a partir de 1..2 son fbf's de L'

Semántica

Para precisar la semántica asociada al lenguaje L' tenemos que determinar cómo interpretamos sus fbf's en el modelo constituido por la extensión de la BDOO.

Definición 4.6 *Una interpretación I de un lenguaje de primer orden L consiste en:*

1) por cada clase d, un conjunto no vacío D_d llamado dominio de la clase d de la interpretación.

2) por cada constante de clase d, una asignación a un elemento de D_d

3) para cada símbolo de predicado n-ario de clase $d_1, \ldots, d_n$ de L una asignación de una relación sobre $D_1, \ldots, D_n$.

Tomamos como dominio de la interpretación los tipos de los identificadores de objetos y los tipos de las clases primitivas (dominios) e introducimos un etiquetado temporal, siendo el tiempo el último argumento en la aridad de eventos y atributos.

Sólo dos construcciones necesitan una explicación adicional: los símbolos de predicado que denotan existencia de instancias de objetos y los eventos relevantes

* sea λ la función de denotación semántica, y sea $C(v_1, \ldots, v_n)$ una instancia de una clase C cuyos eventos de creación y destrucción de instancias son $new_c, destroy_c$ respectivamente, entonces:

$$\lambda(C(v_1,\ldots,v_n,t)) = true\ if\ existe(C(v_1,\ldots,v_r,t))\ \text{else false}$$

donde **existe** se define como se indica a continuación:

$$existe(C(v_1,\ldots,v_r,t)\ if\ new_c(C(v_1,\ldots,v_r,t1)),t1 < t, not(destroy_c(C(v_1,\ldots,v_r,t2)),t2 < t,t2 > t1)$$

* sea $e(v_1,...,v_n)$ una instancia de un evento e. Entonces,

$$\lambda(e(v_1,\ldots,v_r,t)) = true\ if\ ocurre(e(v_1,...v_r,t)\ and\ precond(e(v_1,...,v_r,t)) = true\ else\ false$$

donde **ocurre**$(e(v_1,\ldots,v_r,t))$ significa que el entorno o la actividad interna del Sistema activan el evento e, y **precond** es la precondición asociada al evento.

Acabamos este apartado mostrando algunos ejemplos de consultas que se pueden realizar:

* La consulta **libro(b,_,t),prestar(b,'s1',t),t<10** devolvería los libros que han sido prestados al socio 's1' antes del instante temporal etiquetado como 10.

* Si queremos conocer los socios que tiene actualmente la Biblioteca, suponiendo que el instante actual es el t=20, escribiremos: **socio(s,_,20)**

* Los libros que han sido prestados y no devueltos indicando el *socio* infractor se obtendrían a través de la siguiente consulta: **prestar(b,s,t),not (devolver(b,s,t'),t<t')**

4.2.3 Expresividad lógica de primer orden sin restricciones

La expresividad lógica de la versión clausal de Oasis es, como su propio nombre indica, clausal normal. Es decir, está restringida al uso de cláusulas normales [77] (aquellas compuestas por literales positivos o

negativos) y definidas (con un solo átomo en cabeza), desaprovechando así el poder expresivo completo de las fórmulas de primer orden generales.

La introducción de *sentencias de programa* en la expresividad clausal de Oasis va a proporcionar esa mayor potencia expresiva buscada.

Definición 4.7 *Una sentencia de programa se define como una fórmula de primer orden de la forma*

$$A \leftarrow W$$

donde A es un átomo y W es una fórmula de primer orden no necesariamente cerrada. La fórmula W puede no aparecer. Todas las variables de A y las variables libres de W se asumen universalmente cuantificadas en cabeza de la sentencia de programa. A es la cabeza de la sentencia de programa, y W el cuerpo.

El enriquecimiento de la expresividad de C-Oasis a través de la utilización de sentencias de programa, permite disponer de un lenguaje de especificación con una expresividad de primer orden completa. Tal versión de Oasis recibe el nombre de C^+-Oasis, y ha sido implementada dentro del entorno de producción de Software presentado [141].

El lenguaje asociado a la Teoría de primer orden generada admite en C^+-Oasis un nuevo tipo de fbf's, generadas utilizando cuantificadores existenciales o universales dependiéndo del tipo de definición que se esté realizando:

* Si F es una fbf's clasificada y x es una variable de clase C, $\forall_C x F$ y $\exists_C x F$ son fbf's

El marco formal es el mismo presentado en los apartados anteriores, con esa importante salvedad: el uso de sentencias de programa en lugar de las cláusulas normales.

Por ejemplo, en C^+-Oasis, la especificación del atributo *número* de libros de la clase *lector* se hace como se indica a continuación:

```
numero(r,n,t) if ∃n',∃t1 (t1 is t-1 and numero(r,n',t1)) and
                 ((prestar(b,r,t) and n is n'+1) or
                 (devolver(b,r,t) and n is n'-1))
```

La introducción del cuantificador existencial para las variables n', t_1 permite expresar con precisión lo que se está definiendo.

La Teoría General de Primer Orden generada se usa como esquema de definición de una Base de Datos Deductiva en el prototipo implementado en [26]. Este prototipo normaliza la Teoría General resultante a través de una implementación del Algoritmo de Lloyd [77], que produce el programa Lógico Prolog equivalente a la especificación inicial, manipulado como una Base de Datos Deductiva (BDD). La configuración de este entorno de trabajo se muestra en la figura 4.1 En [141] se muestra (entre otros ejemplos) la definición de la BDD equivalente a la especificación de la biblioteca que venimos usando como ejemplo, que es la fijada por el prototipo de BDD comentado. Esta aproximación conecta el mundo de las especificaciones OO en O^3 con el mundo de las Bases de Datos Deductivas, en la linea marcada en esta Tesis como objetivo : diseñar y desarrollar modernos entornos de producción automática de Software.

4.3 Entorno funcional:F-Oasis

Con la versión funcional de Oasis vamos a proceder de forma similar. La Teoría objeto para una especificación en F-Oasis va a darse como una presentación en un lenguaje de programación funcional como Axis [29] o RAP [66]. En esta Teoría funcional de primer orden resultante las clases se representarán mediante géneros ('sorts'), y sus atributos y eventos serán operaciones consultoras y constructoras respectivamente.

El correspondiente lenguaje de usuario se define de forma similar a como lo hemos hecho en el caso clausal, teniendo presente que ahora sólo dispondremos de un predicado: el de igualdad. Por tanto, las fbf's del lenguaje serán conjuntos de ecuaciones (con variables lógicas si el lenguaje funcional objeto las soporta).

La semántica declarativa asociada a una especificación en F-Oasis viene caracterizada por el modelo inicial de la teoría ecuacional equivalente a la especificación.

La semántica operacional se puede implementar de diferentes formas dependiendo del mecanismo utilizado por el lenguaje funcional objeto en el entorno de trabajo seleccionado:

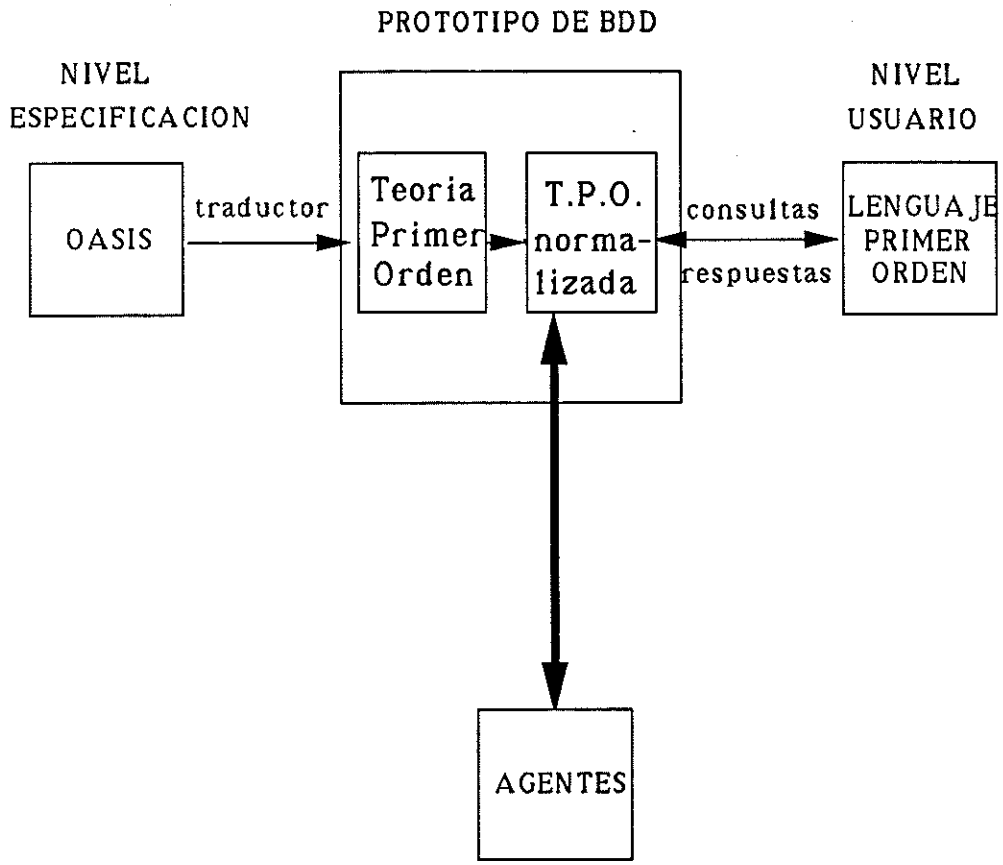


Figura 4.1: Representación gráfica de la configuración del entorno de trabajo que genera una BDD equivalente a una especificación Oasis.

* reescritura (condicional) de términos (Axis).
* 'narrowing' condicional (RAP) .
* resolución SLD-plana ('flat'-SLD) (Europa)

4.3.1 Teoría Ecuacional de Primer Orden asociada a una especificación en F-Oasis

Esquemáticamente, un *álgebra heterogenea ('many-sorted algebra')* [64] es una estructura compuesta por

* una familia de conjuntos de objetos
* un conjunto de funciones con argumentos y resultados pertenecientes a esos conjuntos

Para hacer posible la comunicación, una descripción de un álgebra es necesaria. Esa descripción recibe el nombre de **signatura**. Una signatura introduce nombres para los conjuntos de objetos del álgebra (el género o 'sort' de los respectivos conjuntos). Además, da nombre a objetos y funciones del álgebra a través de las operaciones 0-arias y n-arias respectivamente.

Si a una signatura le añadimos un conjunto de variables y un conjunto de axiomas ecuacionales, tenemos una **presentación.**

Por tanto, para caracterizar la presentación algebraica equivalente a una especificación en F-Oasis, tenemos que determinar su signatura y sus axiomas asociados.

Definición 4.8 *La presentación algebraica correspondiente a una especificación F-Oasis se compone de:*

* *una signatura formada por*
  1) *un género por cada clase de la especificación.*
  2) *un género con el nombre de la especificación, que es el 'género' de interés cuyo tipo es el conjunto de trazas del Sistema.*

* *un conjunto de operaciones, formado por la unión de operaciones constructoras y operaciones consultoras.*

  El género de interés se añade como último género en la aridad de todo operador, sea consultor o constructor.

  1) *los* **eventos** *de las clases van a ser operadores constructores, y su codominio es el género de interés.*
  2) *los* **atributos** *van a ser operadores consultores, cuyo codominio es la clase del atributo.*

* *un conjunto de variables, procedentes de la especificación fuente, al que se le añade una variable de clase 'género' de interés.*

* *un conjunto de axiomas, construidos a partir de los axiomas ecuacionales con los que se definen los atributos variables en la especificación fuente*

Vamos a ilustrar con un ejemplo la conversión de la definición de un atributo variable en F-Oasis en los correspondientes axiomas propios del consultor a que da lugar en la presentación algebraica equivalente. Sea por ejemplo el atributo *número* de libros de la clase *socio*. En F-Oasis se define como:

```
numero(socio):nat (0)
 equations: r:socio;b:libro;n:nat;t:time;
  numero(r)=n if T=T1'prestar(b,r,t) and numero(r)=n-1 at T1.
  numero(r)=n if T=T1'devolver(b,r,t) and numero(r)=n+1 at T1.
              else numero(r) at T1.
```

Esta declaración genera la definición de la operación *número* y sus correspondientes axiomas en la Teoría ecuacional equivalente a la especificación:

```
numero:socio biblioteca ----> nat

vars r:socio,t:tiempo,tr:biblioteca

AXIOMS for numero:
```

```
numero(r,inscribir(r,t,tr))=0
numero(r,prestar(b,r,t,tr))=numero(r,tr)+1
numero(r,devolver(b,r,t,tr))=numero(r,tr)-1
numero(r,evento(...,tr))=numero(r,tr)
```

Es interesante resaltar lo siguiente:

1) tr es una variable del género asociado a la Especificación, que representa trazas o ciclos de vida del Sistema objeto. Es el género de interés y se introduce como último argumento de eventos y atributos tal y como se ha indicado (en nuestro caso, *biblioteca*).

2) El primer axioma es el de inicialización: al crear el objeto, el atributo toma el valor declarado por defecto para el atributo que en este caso es 0.

3) Los dos axiomas siguientes hacen referencia a los eventos relevantes para el atributo considerado, dando cuenta ecuacionalmente de cómo modifican su valor. En el capítulo 6 veremos que los atributos se pueden categorizar, y que la detección de la categoría correspondiente a un atributo permite automatizar su definición.

4) Finalmente, el último axioma es una regla marco general. *Evento* representa cualquier otro evento de la clase, no relevante para el atributo que se está definiendo. Por cada evento no relevante, hay que introducir un axioma que indica que el valor del atributo no cambia ante ocurrencias del evento considerado.

Precondiciones y relaciones de disparo van a ser gestionadas por el animador, encargado de controlar la evolución dinámica de la Teoría Ecuacional equivalente a la especificación. El estado del sistema está representado en todo momento por un término constante del género de interés que refleja la vida del sistema a través del conjunto de eventos ocurridos.

En [109] se presenta una especificación pasiva en F-Oasis del SO Biblioteca, y su correspondiente presentación algebraica en RAP. El mismo ejemplo para un entorno de programación en el que el lenguaje funcional objeto es Axis se presenta en [67].

4.3.2 Lenguaje de Primer Orden asociado a una especificación en F-Oasis como lenguaje de usuario

Una característica importante de un entorno de prototipación es la existencia de un Lenguaje de Usuario que permita a usuarios finales consultar la Teoría de Primer Orden generada equivalente a la especificación. A través de un lenguaje tal se puede conocer, por ejemplo, el valor de los atributos de un objeto en un estado determinado, la extensión de una clase, etc.

Vamos a definir ahora la sintaxis y semántica asociadas a este lenguaje:

Sintaxis

Definición 4.9 *Los términos del lenguaje L asociado a la presentación algebraica obtenida a partir de una especificación F-Oasis se generan inductivamente como se indica a continuación:*

1) *Dado el conjunto de géneros* $\{S_1, \ldots, S_n\}$, *para cada* S_i *existe un conjunto único de nombres* $\{X_{i_1}, \ldots, X_{in}\}$ *que será llamado* **conjunto de variables del género** S_i. *Estas variables son términos de tipo* S_i

2) *Toda constante de un género S procedente de una clase primitiva (dominio) es un término de tipo S*

3) *Sea* S_C *un género que representa a una clase C, y sea* S_A *el género del atributo clave de la clase C. Entonces existe una función* $I_C : S_A \rightarrow S_C \mid si\ s \in S_A$ *entonces* $I_C(s)$ *es un término de tipo* S_C.

   Por ejemplo, **libro('El Quijote')** *es un término de clase* libro

4) *Para toda clase, dado el conjunto de operadores n-arios (constituido por sus eventos)* $o_i : S_{i_1}, \ldots, S_{i_n} \rightarrow S_j$ *con* $S_j = S_{i_n}$, *y dados n términos* $t_1, \ldots, t_n$ *de tipo* $S_{i_1}, \ldots, S_{i_n}$ *respectivamente, entonces* $o_i(t_1, \ldots, t_n)$ *es un término de tipo* $S_j\ \forall i$

   Por ejemplo, dado el operador

$$prestar : Libro \times Socio \times Biblioteca \rightarrow Biblioteca$$

prestar(libro('El Quijote'),socio(254),adquirir(libro('El Quijote'), inscribir(socio(254),nil)))

es un término de tipo Biblioteca

5) *Sea f un operador n-ario correspondiente a un atributo A de una clase C. Sea S_A el género que representa el tipo del atributo, y sean $t_1, \ldots, t_n$ términos del tipo de sus argumentos. Entonces, $f(t_1, ..., t_n)$ es un término de tipo S_A*

Como ejemplo, el término

disponible(libro('El Quijote'),prestar(libro('El Quijote'),socio(254),adquirir(libro('El Quijote'), inscribir(socio(254),nil))))

es de tipo booleano, ya que disponible *es un atributo de clase booleana.*

6) *Los únicos términos de L son los obtenidos aplicando las reglas anteriores.*

Definición 4.10 *Las* **fbf's** *de L se definen inductivamente de la siguiente forma:*

1) *Si t_1, t_2 son términos de L del mismo género, entonces $t_1 = t_2$ es una fbf de L.*

2) *Sólo las generadas a partir de 1) son fbf's.*

Semántica

En este caso, tomamos los términos del género que representa al SO como género de interés, como dominio de la interpretación. Su tipo es el conjunto de trazas o ciclos de vida posibles del Sistema. El estado del Sistema se representa como un término de dicho género, construido

a partir de los operadores constructores (los eventos) y consultado haciendo uso de los operadores consultores (los atributos) y sus axiomas para obtener los valores de dichos atributos en un estado dado, a través del mecanismo deductivo utilizado.

Para estudiar algunos ejemplos de fbf's de L, supongamos que el Sistema se encuentra en un estado en el que se han adquirido dos libros *b1,b2*, se han dado de alta dos socios *s1,s2*, se ha prestado el libro *b1* al socio *s2*, se ha prestado el libro *b2* al socio *s1*, y se ha producido la devolución de este último préstamo. Es decir, representamos el estado de la biblioteca con el siguiente término t de género *biblioteca*:

τ=devolver('b2','r1',prestar('b2,'r1',prestar('b1','r2',inscribir('r2',inscribir('r1',
adquirir('b2',adquirir('b1,nil)))))))

Ejemplos de fbf's en este contexto son:

ECUACION	VALOR
libro(b1,τ)=false	False
libro(b2,τ)=true	True
disponible(b1,τ)=true	True
disponible(b2,τ)=true	False
préstamo(b1,r1,τ)=false	true
libro(B,τ)=true	True if B=b1 or B=b2
libro(b1,τ)=V	True if V=true
préstamo(b1,S,τ)=true	True if S=s2
disponible(B,τ)=true	True if B=b1
libro(B,τ)=false	Sin solución

Las ecuaciones 1..5 son básicas, y son el único tipo de formulas aceptado por un sistema de reescritura de términos. Es el caso del entorno de prototipación que utiliza Axis como lenguaje objeto.

La selección de un lenguaje objeto que soporte el uso de variables lógicas (como es el caso de RAP, con semántica operacional por 'narrowing' condicional), permite escribir ecuaciones más ricas, como las que aparecen en la tabla anterior (6..10). Podemos preguntar por los libros presentes en la Biblioteca en el estado representado por la traza τ (ejemplo 6), por los libros que están disponibles en dicho estado (ejemplo 9),

etc. En estos casos, términos que empiezan con letras mayúsculas son variables, y el intérprete busca sustituciones para estas variables que hagan ciertas las ecuaciones.

Al presentar en el capítulo 5 OO-Method como una Metodología de Producción Automática de Software que cubre todas las fases clásicas en el desarrollo de Software (Análisis, Diseño e Implementación), pero desde una perspectiva OO y deductiva, veremos como las Teorías de Primer Orden equivalentes a una especificación Oasis encajan en el entorno presentado. Constituyen el soporte formal que permite obtener una implementación lógica equivalente a la especificación resultante del Diseño, proporcionando un entorno de trabajo que implementa el Paradigma de Programación Automática a través del uso de métodos formales en todas las fases del Desarrollo.

4.4 Entorno clausal con igualdad:L-Oasis

Una especificación en L-Oasis es equivalente a una teoría clausal con igualdad. Como en los casos anteriores, un traductor construirá, a partir de la especificación, las correspondientes cláusulas de Horn con igualdad. El lenguaje de primer orden asociado a esa teoría se utiliza como base del lenguaje de usuario que permitirá la realización de consultas.

Al existir lenguajes de programación que manipulan la teoría formal equivalente, podremos ejecutar las especificaciones, obteniendo un prototipo del Sistema. En nuestro caso, un programa Europa ([4]) representará la teoría formal subyacente.

Europa es un lenguaje de programación desarrollado en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. Está diseñado para integrar el paradigma clausal y el paradigma ecuacional, añadiendo la igualdad a un lenguaje lógico para obtener un lenguaje lógico con características funcionales.

Desde el punto de vista sintáctico, Europa utiliza una notación funcional sin tipos ni funciones de alto nivel y sigue una disciplina de constructores. Sus programas son un conjunto de cláusulas de Horn con igualdad.

Su semántica declarativa está definida como un refinamiento del

modelo mínimo de Herbrand. La semántica operacional está basada en resolución SLD-plana ('Flat-SLD Resolution'), construida añadiendo a la semántica operacional de Prolog un proceso inicial de aplanamiento ('flattening').

Este entorno de prototipación automática se presenta con detalle en [37].

4.4.1 La Teoría Clausal con Igualdad Equivalente

La teoría objeto, que un traductor construirá a partir de la especificación L-Oasis, está formada por cláusulas de Horn con igualdad. El conjunto de dichas cláusulas formará el programa Europa correspondiente a dicha especificación.

En el paso desde la especificación L-Oasis al programa Europa equivalente, se da la siguiente correspondencia:

* evento → constructor
* evento destrucción → función
* atributo → función
* clase → predicado

A nivel de la teoría formal, se tendrán axiomas para la existencia de una instancia de clase (predicado) y para conocer el valor de los atributos (función). No se darán axiomas entre constructores. Los eventos de destrucción no son constructores sino funciones.[3] Un monitor de

[3]Para este tipo particular de eventos, L-Oasis funciona con una aproximación conceptualmente dinámica, pues el efecto de su ocurrencia no es el almacenamiento del evento (como lo ha sido hasta ahora), sino que lo que devuelve la función asociada al evento de destrucción es una nueva traza en la que el objeto destruido no aparece. No obstante, el tratamiento para el resto de los eventos y para el Sistema en general es deductivo en lo que se refiere a objetos existentes, para los que el valor de sus atributos en el estado considerado se deduce de los eventos que constituyen la traza o ciclo de vida que representa la vida del objeto.

Aunque esta tesis se centra en la vocación deductiva de Oasis, la representación de eventos a través de funciones posibilita el tratamiento de especificaciones en Oasis como Sistemas Dinámicos, con lo que una especificación en Oasis podría interpretarse como deductiva o dinámica segun se desease. Este es un punto interesante en el que se va a continuar trabajando en un futuro inmediato.

comportamiento garantiza que los eventos introducidos en el Sistema llevan a una traza admisible, en el contexto de la Animación de la Especificicación que se presenta con detalle en la siguiente sección.

En dicho monitor de comportamiento se incluyen las precondiciones de la especificación, de forma adecuada para preguntar a la Teoría Formal base. Cuando un evento es activado, el monitor comprobará su precondición interactuando con la teoría. Si no se satisface, no se toma ninguna acción; en caso contrario, se inserta el nuevo evento en la traza.

La información sobre disparos se almacena también para la correcta animación de la especificación. De forma análoga, se incluirán de forma adecuada para preguntar a la teoría formal por las condiciones especificadas. El animador comprobará en cada momento la posibilidad del disparo.

Veamos cómo se obtiene la teoría formal de la que estamos hablando:

4.4.2 Teoría Clausal de Primer Orden con Igualdad asociada a una especificación en L-Oasis

Vamos a ver en primer lugar el **Lenguaje de Primer Orden** de la Teoría Clausal con igualdad correspondiente a una especificación en L-Oasis.

El lenguaje L de primer orden de la Teoría Clausal con igualdad clasificada asociada a una especificación en L-Oasis se define inductivamente de la siguiente forma, a partir de un alfabeto compuesto por variables, constantes, símbolos de función y símbolos de predicado (todos ellos clasificados) proporcionados por la especificación fuente en Oasis:

Definición 4.11 *Los* **términos** *de clase d se definen inductivamente como:*

1) Una constante de clase d es un término de clase d

2) Una variable de clase d es un término de clase d

3) Sea e el nombre de un evento de aridad n con $e : S_1, \ldots, S_n \rightarrow S_j$ *con* $S_j = S_n$*, y sean* $t_1, \ldots, t_n$ *términos de las clases correspon-*

dientes. Entonces $e(t_1, \ldots, t_n)$ *es un término de clase* S_j *(que representa una traza de eventos).*

4) *Sea a el nombre de un atributo de clase d que pertenece a una clase c; sea k el valor de los atributos clave de una de las instancias de c y sea t una traza de eventos. Entonces* $a(k,t)$ *es un término de clase d.*

5) *Solo los términos obtenidos a partir de las reglas anteriores son de clase d.*

Definamos ahora los **átomos** del lenguaje de primer orden:

Definición 4.12 *Los átomos del lenguaje L se definen inductivamente como:*

1) *Si C es una clase con atributos clave* $a_1, ..., a_n$ *de clase* $C_1, ..., C_n$, $t_1, ..., t_n$ *son términos de clase* $C_1, ..., C_n$ *respectivamente y t es una traza de eventos de C, entonces* $C(t_1, ..., t_n, t)$ *es un átomo* **relacional** *que representa la existencia de la instancia de la clase C con clave* $a_1, ..., a_n$. *Por ejemplo, podemos citar:*

$$socio('s1', t)$$

2) *Sea a el nombre de un atributo de clase d perteneciente a una clase C, v un término de clase d, k el valor de los atributos clave de una instancia de C y t una traza de eventos; entonces* $= (a(k,t), v)$ *es un átomo* **funcional** *que representa el valor del atributo a. Utilizaremos la notación infija habitual y escribiremos:*

$$a(k,t) = v$$

Como ejemplo de un átomo de este tipo, tenemos:

$$disponible('El\ Quijote', prestar('El\ Quijote', socio(254), adquirir('El\ Quijote', inscribir(socio(254), nil)))) = false$$

3) Dado el conjunto ops=$\{<, >, \leq, \geq\}$ de operadores relacionales, si f,g son términos de la misma clase (primitiva) y para dicha clase los operadores ops están definidos, entonces f **op** *g es un átomo si op $\in$ ops.*

Estos átomos permiten efectuar sencillas relaciones entre atributos y/o valores de atributos. Por ejemplo,

num_libros(n,t)<10.

4) Sólo son átomos de L los obtenidos a través de las reglas anteriores.

Sólo falta ahora determinar las fbf's de L.

Definición 4.13 *Las fbf's de L se definen inductivamente de la siguiente manera:*

1) Un átomo de L es una fbf.

2) Si F,G son fbf's, entonces not(F), F and G, F or G, son fbf's.

3) Si A es un átomo funcional o relacional, y B es una fbf, entonces $A \leftarrow B$ es una fbf.

4) Sólo son fbf's de L las obtenidas a partir de las dos reglas anteriores.

Definido el lenguaje de primer orden de la Teoría equivalente a la especificación, vamos a ver cuales van a ser sus axiomas:

Definición 4.14 *La Teoría clausal con igualdad equivalente a una especificación en L-Oasis va a contener los siguientes tipos de axiomas:*

* **Constructores, funciones y predicados.** *Los* **constructores** *serán los eventos, a excepción de los eventos de destrucción especificados en las clases que no son constructores sino funciones. La aridad de cada constructor es $(N-1)+1$, donde N es la aridad del evento correspondiente, es decir:*

- $N-1$:*un argumento por cada argumento del evento, excepto el último de tipo tiempo.*
- *un argumento adicional que representa la traza de eventos.*

Los eventos de creación tienen un argumento más por cada atributo constante de la clase, ya que es en el momento de la creación de una instancia cuando se dan valores a los atributos constantes correspondientes. El constructor nil representa una traza de eventos vacía (estado inicial).

Las **funciones** *son los eventos de destrucción y los atributos. Los eventos de destrucción tienen como resultado una traza de eventos. El resultado de los atributos constantes es constante y se da al crear un objeto. El valor de los atributos variables viene dado por su definición axiomática en la especificación (función de observación).*

Los **predicados** *son las clases, y tendrán éxito si una determinada instancia existe.*

* **Axiomas para clases.** *Cada clase de una especificación se representa como un predicado en la teoría formal de primer orden con igualdad equivalente, con axiomas que caracterizan la existencia de instancias de dicha clase. Para cada constructor se dan axiomas de la clase como se indica a continuación:*

1) *clase(C1,new_evento(C2,T):-C1=C2.*
2) *clase(C1,new_evento(C2,T):-no(C1=C2),clase(C1,T).*
3) *clase(C1,otro_evento(C2,T):-clase(C1,T).*

siendo clase *el nombre de la clase,* new_evento *el evento creador de instancias de la clase considerada, y* otro_evento *cualquier constructor (evento) distinto al evento de creación.*

En el caso de la biblioteca, tendremos por ejemplo los axiomas siguientes para la clase libro*:*

- *libro(Libro,adquirir(L,Titulo,Autor)):-Libro=L.*

- *libro(Libro,adquirir(L,Titulo,Fecha,Autor)):-no(Libro=L), libro(Libro,T).*
- *libro(Libro,prestar(L,S,T)):-libro(Libro,T).*
- *libro(Libro,devolver(L,S,T)):-libro(Libro,T).*
- *libro(Libro,nil)=false*

El evento relevante es el de creación correspondiente. Para los restantes se busca en el estado anterior. Una instancia de clase no existe si no se encuentra el evento de creación en la traza.

* **Axiomas para los eventos de destrucción.** *Como los objetos dejan de existir ante la ocurrencia de sus eventos de destrucción, habrá axiomas para éstos que anulen el efecto de los correspondientes eventos de creación. Es decir, un evento de destrucción tras el correspondiente evento de creación deja la traza de eventos como estaba. Los eventos de destrucción se representan por tanto como funciones, cuyo resultado será una traza de eventos:*

- *destroy_evento(C1,new_evento(C2,T))=T :- C1=C2.*
- *destroy_evento(C1,new_evento(C2,T)) = new_evento(C2,destroy_evento(C1,T)) :- no(C1=C2).*
- *destroy_evento(C1,otro_evento(C2,T)) = otro_evento(C2,destroy_evento(C1,T)).*

donde destroy_evento *es el evento destrucción de instancias de la clase considerada,* new_evento *el de creación, y* otro_evento *cualquier constructor (evento) distinto al evento de creación.*

El evento de destrucción se va desplazando a lo largo de la traza de eventos hasta que se encuentre el correpondiente evento de creación. En nuestro ejemplo, para la clase libro *tendremos:*

- *eliminar(Libro,adquirir(L,Titulo,Fecha,Autor,T))=T:-Libro=L.*
- *eliminar(L1,adquirir(L,Titulo,Fecha,Autor,T)) = adquirir(L,Titulo,Fecha,Autor,eliminar(L1,T)) :-no (L1=L).*
- *eliminar(L1,prestar(L,S,T))=prestar(L,S,eliminar(L1,T)).*

– *eliminar(L1,devolver(L,S,T))=devolver(L,S,eliminar(L1,T)).*

* **Axiomas para atributos variables.** *En una especificación L-Oasis, la forma en la que los atributos variables toman diferentes valores aparece explícitamente en las fórmulas de su especificación. Lo único que hay que hacer es traducir dichas fórmulas a la notación de trazas que estamos usando, tranformando cada atributo en una función que toma un valor según la traza de eventos ocurrida. De esta manera, la fórmula*

$$atributo(C,T) = valor :- evento(C,T).$$

genera los siguientes axiomas:

1) atributo(C1,evento(C2,T))=valor :- C1=C2.

2) atributo(C1,evento(C2,T))=atributo(C1,T):- no(C1=C2).

Por defecto, cuando en la especificación L-Oasis de un atributo variable no aparece una fórmula para un determinado evento, se entiende que dicho evento no es relevante para el atributo (regla marco). Por ello, en la Teoría formal que estamos elaborando, la definición del atributo variable se completa con axiomas para el resto de constructores, que den cuenta de esta regla marco:

$$atributo(C1,otro\_evento(C2,T))=atributo(C1,T).$$

No hay que incluir axiomas para el evento de destrucción porque como hemos visto éste no es un constructor. Cuando se desee conocer el valor de un atributo de un objeto, un monitor de comportamiento (el animador de la especificación que veremos en la sección siguiente) debe comprobar que la instancia de clase consultada existe antes de calcular su valor.

En el caso de la clase libro *de la Biblioteca que venimos usando como ejemplo, tendremos por ejemplo la siguiente definición para el atributo variable* disponible*:*

– *disponible(L1,adquirir(L,Titulo,Fecha,Autor,T))=true:-L1=L.*

- *disponible(L1,adquirir(L,Titulo,Fecha,Autor,T))=disponible(L1,T) :- no(L1=L).*
- *disponible(L1,prestar(L,S,T))=false:-L1=L.*
- *disponible(L1,prestar(L,S,T))=disponible(L1,T):-no(L1=L).*
- *disponible(L1,devolver(L,S,T))=true:-L1=L.*
- *disponible(L1,devolver(L,S,T))=disponible(L1,T):-no(L1=L).*

* **Axiomas para atributos constantes**. *Para hacer homogeneo el tratamiento de todos los atributos de una clase, los atributos constantes serán funciones que van a ser modificadas por ciertos eventos particulares: los de creación de instancias. En el momento de la creación de un objeto es cuando se dan valores a sus atributos constantes.*

Como ejemplo, en la clase libro *tenemos un atributo constante* título, *cuya definición sería la siguiente:*

- *titulo(Libro,adquirir(L,Titulo,Fecha,Autor,T))=Titulo:-Libro=L.*
- *titulo(Libro,adquirir(L,Titulo,Fecha,Autor,T))=titulo(Libro,T):-no(Libro=L).*
- *titulo(Libro,prestar(L,S,T))=titulo(Libro,T)*
- *titulo(Libro,devolver(L,S,T))=titulo(Libro,T)*

El evento de creación de instancias de la clase libro, adquirir, *está representado en la teoría por el constructor*

adquirir(L,Titulo,Fecha,Autor,T)

. *El argumento L representa la clave de la clase* libro, *los argumentos* Título, Fecha y Autor *representan sus atributos constantes y el argumento T representa el resto de la traza de eventos. El atributo constante* titulo *es una función cuyo valor viene dado por el argumento correspondiente del constructor (evento de creación).*

* **Axiomas de precondiciones y de relaciones de disparo.** *A partir de las precondiciones y los disparos de la especificación L-Oasis fuente, se construyen cláusulas cuyos cuerpos los constituyen las condiciones correspondientes, traducidas de manera adecuada para preguntar a la Teoría en fase de animación si la condición considerada se satisface.*

  Estos axiomas van a caracterizar las modificaciones de la Teoría de Primer Orden con igualdad resultante de la especificación inicial.

En [37] se muestra detalladamente como una especificación en L-Oasis es equivalente a un programa lógico, representando su teoría clausal con igualdad subyacente como un programa Europa.

El traductor allí implementado produce el programa Europa correspondiente a una especificación dada, permitiéndonos hacerla ejecutable. Además, construye un fichero de parámetros que contiene precondiciones y disparos debidamente traducidos para hacer posible la animación de la especificación.

4.4.3 Lenguaje de Primer Orden asociado a una especificación en L-Oasis como lenguaje de usuario

Sintaxis

El lenguaje L' que se pone a disposición del usuario para que sea utilizado como lenguaje de consulta es en este caso el mismo lenguaje L, propio de la Teoría de primer orden con igualdad equivalente a la especificación.

Semántica

La semántica para la teoría formal correspondiente a una especificación L-Oasis es una combinación de los aspectos clausales y ecuacionales vistos en las secciones anteriores.

Por una parte, viene dada tomando los tipos de los identificadores de objetos y los tipos de los dominios (clases primitivas) como dominio

de la interpretación. Por otra, el estado del sistema se representa como un término del género que representa al SO en su conjunto, y cuyo tipo es el conjunto de trazas o ciclos de vida posibles del Sistema. Estos términos sustituyen al etiquetado temporal utilizado en la teoría de primer orden clausal equivalente a una especificación en C-Oasis. Se construyen a partir de los operadores constructores correspondientes, que hemos visto que eran los eventos (excepto el de destrucción).

Cualquier fbf será resuelta utilizando la teoría formal equivalente. El animador garantizará su consistencia comprobando las precondiciones y la existencia de las instancias de clase antes de introducir nuevos eventos en la teoría, y disparando los eventos cuya condición de disparo sea satisfecha.

Veamos finalmente algunos ejemplos de fórmulas bien formadas:

1) libro(l1,adquirir(l2,'El Quijote',2,'Cervantes',adquirir(l1,'Hamlet',1,'Shakespeare',nil))).

   El ejemplo anterior realiza la pregunta sobre la existencia de la instancia de la clase libro con clave *l1* en el estado representado por la traza de eventos dada.

2) disponible(l1,prestar(l1,s1,inscribir(s1,'Carlos',adquirir(l2,'El Quijote',2,'Cervantes',adquirir(l1,'Hamlet',1,'Shakespeare',nil))))=X.

   En este ejemplo se pregunta si el libro con clave *l1* está disponible en el estado dado por la traza de eventos correspondiente.

4.5 Animación de una especificación

En el Programa Lógico (clausal, ecuacional o clausal con igualdad) correspondiente a una especificación en Oasis que se genera automáticamente a partir de dicha especificación, *nil* es la constante del género de interés en el caso ecuacional, o la lista vacía en el caso clausal, que representa la traza en el estado inicial del Sistema. A partir de esta traza inicial, el estado del Sistema irá evolucionado en el tiempo a través de la ocurrencia de eventos.

Animar o **ejecutar** una especificación consiste en dar cuenta del comportamiento dinámico de la Teoría de primer orden correspondiente a la especificación. Cuando un evento se activa, si el evento es relevante hay que modificar dicha Teoría introduciendo el evento en la traza que representa la vida del Sistema. También tiene que ser posible efectuar observaciones o consultas, que permitan obtener información sobre el estado del Sistema a través de los valores de los atributos de los objetos en el estado considerado. Ya hemos definido potentes lenguajes formales de usuario que hacen posible esas consultas.

Para formalizar estos aspectos dinámicos asociados a toda especificación vamos a utilizar dos aproximaciones ortogonales, que comparten como característica esencial el hecho que permiten dar cuenta de una manera formal, sencilla y elegante de la animación de toda especificación. Estas dos aproximaciones son:

* **metaprogramación**, definiendo el Metaprograma Lógico formalmente equivalente a una especificación y dando cuenta de su evolución a través de metapredicados especialmente definidos para ello.

* **manipulación de operadores algebraicos**, dentro del marco formal algebraico desarrollado en el capítulo 2. Estos operadores algebraicos, (eventos) serán constructores de términos que representan estados del sistema. La evaluación de estos términos constituye las observaciones.

En el primer caso, esa 'introducción del evento en la traza' citada anteriormente se representa como la inserción del átomo que representa al evento en la lista correspondiente a la traza o ciclo de vida. De esta manera, la Teoría Clausal equivalente a la especificación capta el conocimiento derivado de la ocurrencia de un evento.

En el segundo, a partir del término que representa el estado actual del Sistema, la ocurrencia de un evento (operador constructor) produce un nuevo término representante del nuevo estado.

Vamos a desarrollar a continuación estas dos aproximaciones:

4.5.1 Metaprogramación

Conceptos básicos relacionados con el uso de técnicas de metaprogramación en Programación Lógica pueden encontrarse en [61,62], de dónde provienen las nociones básicas utilizadas en esta sección.

Un metaprograma es un programa que utiliza otro programa (el programa objeto) como dato. El uso de técnicas de metaprogramación es especialmente útil, como vamos a ver, para expresar propiedades dinámicas de Programas Lógicos.

En [62] el uso de la metaprogramación se restringe al caso estático (programas objeto estáticos). Sin embargo, a menudo se hace necesaria la manipulación dinámica de programas objeto, creando nuevas representaciones de los mismos. La idea fundamental, procedente de Bowen y Kowalski [18] es que para poder manipular correctamente programas objeto de forma dinámica, hay que dotarlos de status de primer orden. Para ser más preciso, hay que representar los programas objeto no como metaprogramas, sino como metatérminos (términos en el nivel 'meta').

La adopción de un enfoque de tipo MetaProlog permite introducir dos nuevos predicados *insertar* y *borrar*. Si representamos programas objeto como términos en el nivel meta, podemos definir estos dos metapredicados como:

* **insertar(programa,fórmula,nuevoprograma)**

  donde *nuevoprograma* es el término que representa el programa obtenido añadiendo la sentencia de programa *fórmula* al programa representado como *programa*.

* **borrar(programa,fórmula,nuevoprograma)**

  siendo en este caso *nuevoprograma* el programa resultante de eliminar la sentencia de programa representada por *fórmula* del programa representado por *programa*.

En este contexto, es sencillo dar una definición de estos predicados con una semántica declarativa y operacional bien definidas [61]. Obviamente, el comportamiento de estos predicados es muy distinto del de los famosos *assert,retract*. Las diferencias principales son dos:

1) el uso de los predicados *insertar* y *borrar* en el nivel 'Meta' nos permite describir formalmente lo que significa modificar una Teoría.

2) el hecho de que los efectos de *insertar* y *borrar* se pueden deshacer. Sus efectos son menos permanentes que los de *assert,retract.*

Estos nuevos metapredicados cumplen con todos los requisitos para ser utilizados en modificaciones de bases de conocimiento en entornos lógicos formales, como lo es el que estamos presentando en este capítulo.

Si una especificación en Oasis es equivalente a una Teoría de primer orden, y si esta teoría se representa mediante un programa lógico, el uso de metaprogramación nos permite dar cuenta de la animación de esa teoría de la siguiente forma:

1) Definiendo el metalenguaje correspondiente a una especificación Oasis: el programa lógico objeto que representa a la especificación en su estado inicial es un término de este metalenguaje.

2) Representando los estados del Sistema con un término del metalenguaje, que llamaremos **historia**. Una historia está formada por la secuencia de eventos ocurridos en el Sistema.

3) Definiendo un metapredicado **insertar** que incluya eventos relevantes en el programa lógico que representa a la Teoría. Se caracteriza formalmente la dinámica de la Sociedad de Objetos definiendo **insertar** con tres argumentos:

   (a) la historia actual del sistema.

   (b) el evento relevante que se quiere insertar.

   (c) la nueva historia del sistema, que es la obtenida añadiendo el evento relevante considerado a la historia inicial.

Dado un estado del sistema representado por una historia, un conjunto de metapredicados predefinidos permitirán efectuar observaciones (obtener los valores de los atributos de cualquier objeto de la Sociedad en ese estado).

Representación base

La representación base es un esquema para representar las fórmulas de un lenguaje L en otro lenguaje L'. Vamos a partir de un lenguaje L no

clasificado para simplificar la presentación, y para exponer claramente la aportación que supone el uso de metaprogramación al problema de caracterizar la dinámica de cualquier Sistema que admita una representación lógica. La extensión al caso que L sea clasificado es inmediata.

Precisemos en primer lugar nuestro lenguaje L de partida. Vamos a definir el lenguaje L no clasificado correspondiente a una especificación en Oasis, de la misma forma en que lo hacíamos para el caso general clasificado en la sección anterior.

Definición 4.15 *Los términos del lenguaje L se definen inductivamente como se indica a continuación:*

1) Un valor de una clase primitiva (dominio) d es un término.

2) Una variable de una clase primitiva (dominio) d es un término.

3) Sólo son términos de L los obtenidos a partir de las dos reglas anteriores.

Definición 4.16 *Los átomos de L se definen inductivamente de la siguiente forma:*

1) Si C es una clase con atributos constantes $a_1, \ldots, a_n$, *y* $t_1, \ldots, t_n, t$ *son términos, entonces* $C(t_1, \ldots, t_n, t, t')$ [4] *es un átomo de L.*

2) Si a es un atributo de una clase, y c, v, t *son términos, entonces* $a(c, v, t)$ *es un átomo de L.*

3) Si e es un nombre de evento, y $t_1, \ldots, t_n$ *son términos, entonces* $e(t_1, \ldots, t_n)$ *es un átomo de L*

4) Sólo son átomos de L los obtenidos de acuerdo con las reglas anteriores.

Definición 4.17 *Las fbf's de L se definen inductivamente de la siguiente manera:*

[4] t y t' se van a interpretar como etiquetas temporales que constituyen el intervalo de existencia de un objeto: t representa el instante de creación y t' el instante de destrucción

1) Un átomo de L es una fbf de L.

2) Si A,B son fbf's de L, entonces not(A), A and B, A or B, $A \leftarrow B$ son fbf's de L.

3) Si A es una fbf de L y x es una variable, entonces $\forall x\ A$ y $\exists x\ A$ son fbf's de L.

4) Las únicas fbf's de L son las generadas a partir de las tres reglas anteriores.

Definido el lenguaje base L, vamos a caracterizar su representación base:

Representación base de L

Definición 4.18 *Dada una constante C de L, existe una constante correspondiente C' en L'. Esta función $C \rightarrow C'$ es inyectiva.*

Definición 4.19 *Dado un predicado n-ario p en L, existe un* **símbolo de función** *n-ario correspondiente p' en L'. La función $p \rightarrow p'$ es inyectiva.*

L' contiene además:

* las constantes $empty, nil, 0$
* los símbolos de función:
  1) $\neg$ de aridad 1
  2) $\&, |, if, some, all$,cons[5] de aridad 2
* otras constantes, símbolos de función y símbolos de predicado que especificaremos más adelante

Definición 4.20 *Si C es una constante y p es un símbolo de predicado, representamos C como C' y p como p' en L'.*

[5]Usaremos la notación habitual en listas y denotaremos $cons(x, y)$ como $[x \mid y]$, y *nil* como []

Se define la representación de variables de L de la forma más natural: utilizando una constante en L'. Como no se tienen infinitas variables en L, no es necesario utilizar un símbolo especial de función para representar variables como se propone en [62].

Vamos a completar la definición de los términos de L' caracterizando átomos y fbf's de L como términos en L':

Definición 4.21 *Si* $p(t_1, \ldots, t_n)$ *es un átomo de L, entonces representamos* $p(t_1, \ldots, t_n)$ *con el término* $p'(t'_1, \ldots, t'n)$ *de L'.*

Definición 4.22 *A los términos de L' obtenidos a partir del uso del símbolo de función* cons ([_ | _]) *sobre argumentos (términos en L') que proceden de los átomos de L que representan eventos, se les llama* **historias**.

Se definen del mismo modo las fbf's no vacías de L como términos (símbolos de función) de L':

Definición 4.23 *Si F,G son fbf's no vacías de L que se representan como F',G' en L', y las variables* $x_1, \ldots, x_n$ *se representan como* $x'_1, \ldots, x'_n$ *respectivamente en L', entonces not(F),F and G, F or G,F* $\leftarrow$ *G,* $\forall x_1, \ldots, x_n F, \exists x_1, \ldots, x_n F$ *se representan en L' como* $\neg F, F\&G, F \mid G$, *F if G,all*$([x'_1, \ldots, x'_n], F'), some([x'_1, \ldots, x'_{n'}], F')$. *La fórmula vacía se representa en L' como empty.*

Tenemos ahora definidos los términos del metalenguaje. A continuación vamos a introducir los predicados que usaremos en el nivel 'meta' (metapredicados).

Metaprograma equivalente a una especificación Oasis

El objetivo de este apartado es definir los símbolos de metapredicados lógicos que conforman el metaprograma P equivalente a una especificación Oasis. Basándonos en este metaprograma daremos cuenta formalmente de los aspectos dinámicos relacionados con la animación de una especificación Oasis.

La interpretación de estos metapredicados se basa en la preinterpretación de Herbrand para el lenguaje L' definido anteriormente, enriquecido con los símbolos de predicado que se introducen en esta sección.

El subconjunto de términos constantes del Universo de Herbrand que representa las constantes de L lo constituyen los tipos de los identificadores de objetos, y los tipos de los dominios o clases primitivas de la especificación fuente.

Empezamos con el metapredicado **Class**, que se interpretará como cierto cuando su primer argumento representa una instancia de clase existente en el estado del sistema dado por su historia o secuencia de eventos ocurridos(segundo argumento).

Definición 4.24 *Definición del metapredicado* **Class**: *Si C' es el término de L' correspondiente a un átomo de L que representa la existencia de instancias de una clase C, new_C (respectivamente $destroy_C$) es el término de L' correspondiente a un átomo de L que representa al evento generador (respectivamente destructor) de instancias de la clase C, y H es el término (historia) que representa el estado del sistema como una lista de eventos, entonces se define el metapredicado 'class' de la siguiente forma:*

* $Class(C'(t'_1,\ldots,t'_n,t,t').[new_C(t'_1,\ldots,t'_n,t) \mid H]) \leftarrow .$
* $Class(C'(t'_1,\ldots,t'_n,t,t').[destroy_C(t'_1,\ldots,t'_n,t') \mid H]) \leftarrow$
  $Class(C'(t'_1,\ldots,t'_n,t,t').[H]).$
* $Class(C'(t'_1,\ldots,t'_n,t,t').[evento_1(x,t) \mid H]) \leftarrow$
  $Class(C'(t'_1,\ldots,t'_n,t,t').[H]).$
* $\vdots$
* $Class(C'(t'_1,\ldots,t_n,t,t').[evento_n(x,t) \mid H]) \leftarrow$
  $Class(C'(t'_1,\ldots,t'_n,t,t').[H]).$

La definición instancia el intervalo de existencia del objeto cuando se detecta la ocurrencia de sus eventos de creación y destrucción respectivamente.

Las últimas sentencias representan la regla marco: si el evento consultado en la historia del sistema no es el evento generador de instancias de la clase en cuestión sino que es cualquier otro, continuamos la búsqueda en el resto de la historia.

En el ejemplo de la biblioteca, tenemos para la clase libro:

* $Class(libro(t1,t2,t,t'),[adquirir(t1,t2,t)|H]) \leftarrow .$

* $Class(libro(t1,t2,t,t'),[eliminar(t1,t2,t')|H]) \leftarrow$
  $Class(libro(t1,t2,t,t'),[H]).$

* $Class(libro(t1,t2,t,t'),[X|H]) \leftarrow$
  $Class(libro(t1,t2,t,t'),[H]).$

Detectada la ocurrencia de un evento *adquirir* (creador de instancias de la clase *libro*), se instancia su etiqueta temporal de nacimiento. Si se trata de su evento se destrucción, se instancia la etiqueta temporal correspondiente y se busca en la historia el evento que lo creó para cerrar su intervalo de existencia. Si el evento consultado no es ninguno de los dos anteriores, continua la búsqueda sobre la historia del sistema (regla marco).

El metapredicado **attribute** permite deducir el valor un atributo (representado por el nombre del símbolo de función correspondiente del primer argumento) de una clase sobre el estado del sistema representado por su historia (segundo argumento), según la regla deductiva suministrada por la especificación como definición del atributo. El cuerpo de esta regla deductiva (una fórmula en L) es un término en L' de acuerdo con la representación base de L vista, en base al cual se construye el cuerpo del metapredicado **attribute**.

Definición 4.25 *Definición del metapredicado* **Attribute**: *dado un atributo a de una clase c, sea v un valor del tipo del atributo. Dada una historia H y un evento relevante e (siendo l la lista de sus parámetros y t una etiqueta temporal) para el atributo a, se define* **Attribute** *como:*

* $Attribute(a(c,v,t),[e(c,l,t) \mid H]) \leftarrow$ *regla-deductiva-asociada-a-e* . $Attribute(a(c,v,t),[H]).$

  La regla deductiva asociada al evento procede de la definición del atributo a considerado en la especificación Oasis fuente.

  Para cada atributo variable de una clase, se tendrá una definición como la anterior por evento relevante.

* $Attribute(a(c,v,t),[e'(c,l,t) \mid H]) \leftarrow Attribute(a(c,v,t),[H]).$

  que representa la regla marco, en el caso que e' sea cualquier otro evento no relevante para el atributo a

* $Attribute(a(c, vpd, t), [new_c \mid H]) \leftarrow$.

  que representa la asignación del valor por defecto (vpd) correspondiente al atributo variable 'a' de un objeto de la clase 'c' creado por el evento generador de instancias de 'c' new_c.

Vamos a ver como ejemplo la definición del metapredicado **attribute** referida al atributo variable *número* de libros de la clase *socio*. Siendo los eventos relevantes del atributo *número* los de *prestar* y *devolver*, y conocido su efecto sobre el valor de dicho atributo, tenemos la siguiente definición:

```
Attribute(numero(r,n,t),[prestar(b,r,t)|H]) <---
      n is n1+1 , Attribute(numero(r,n1,t),[H]).

Attribute(numero(r,n,t),[devolver(b,r,t)|H]) <---
      n is n1-1 , Attribute(numero(r,n1,t),[H]).

Attribute(numero(r,n,t),[e(b,r,t)|H]) <---
      n is n1 , Attribute(numero(r,n1,t),[H]).

Attribute(numero(r,0,t),[inscribir(b,r,t)|H]) <---.
```

Las dos primeras cláusulas provienen de los eventos relevantes del atributo considerado (*prestar,devolver*), la tercera es la regla marco y la última representa la asignación del valor por defecto asignado al atributo al crear el objeto.

El metapredicado **Event** se interpreta como cierto cuando sus argumentos representan eventos de la especificación. Se define por extensión a partir de la declaración de eventos realizada para toda clase de la especificación fuente.

Definición 4.26 *El metapredicado* **Event** *se define extensionalmente a partir de la especificación fuente. Para toda clase c con eventos* $e_1, \ldots, e_n$ *se tiene:*

$\mathbf{Event}(e_1(c, v, t)) \ldots \mathbf{Event}(e_n(c, v, t))$

siendo c una variable que representa un nombre de clase, v una variable que representa información adicional (opcional) del evento, y t una variable que representa la etiqueta temporal asociada a una ocurrencia de evento.

En el ejemplo que venimos usando tendríamos:

```
Event(adquirir(l,t)).
Event(prestar(l,s,t)).
Event(devolver(l,s,t)).
...
```

El metapredicado **Precond** se define para asociar una precondición a un evento. Se interpretará como cierto cuando la precondición asociada a su primer argumento (que representa un evento) se satisfaga.

Definición 4.27 *Definición del metapredicado* **Precond**: *dado un evento e de una clase c y una historia H, se define* **Precond** *como:*

Precond(e(c,v,t),H) $\leftarrow$ **precondición**

siendo precondición *una fórmula de L procedente de la declaración de precondiciones efectuada en la especificación Oasis fuente. Esta fórmula de L se transforma en una fórmula atómica de L' utilizando los metapredicados definidos anteriormente.*

Por ejemplo, la precondición asociada por defecto a todo evento creador de objetos de una clase es la no existencia del objeto creado. Esto se expresa de la siguiente forma:

$$Precond(adquirir(l,t)) \leftarrow \neg(Class(libro(l,ac,t1,t2)), t > t1, t < t2)$$

En este contexto, la animación de una especificación entendida como la modificación dinámica de la Teoría de primer orden equivalente, se formaliza definiendo un metapredicado **insert**, definido de la siguiente manera:

Definición 4.28 Animación pasiva de una especificación en Oasis. *Dada una historia h que representa el estado actual del sistema (secuencia de eventos relevantes), y un evento e se define el metapredicado* **insert** *como:*

$$Insert(h,e,[e \mid h]) :- Event(e), Precond(e,H).$$

Insert se interpreta como cierto cuando el primer argumento representa la secuencia de eventos o traza que conforma el estado actual del sistema, el segundo argumento el evento que se pretende activar, y el último argumento representa la nueva historia obtenida por concatenación del evento activado a la traza original h. Esta inserción sólo se producirá si:

1) el evento que se va a activar es un evento del sistema;

2) la precondición asociada a dicho evento se satisface.

Suponiendo que la biblioteca usada como ejemplo se encuentra en un estado elemental compuesto por:

H=[adquirir(Isbn1,Hamlet,2), inscribir(1,Juan,1), nil],

la ocurrencia de un *préstamo* del libro *Hamlet* al socio *Juan* se representa como una instancia del metapredicado **insert**:

Insert(H,prestar(Isbn1,1,3),H').

El evento solo será relevante y será añadido a la historia inicial si se comprueba que es un evento (usando el metapredicado Event) y su precondición se cumple (controlado por el metapredicado Precond).

Se ha visto cómo la definición de relaciones de disparo introduce actividad en la Sociedad de objetos interactivos denotada por una especificación Oasis. A nivel del metalenguaje, se introduce un metapredicado **trigger**, que se interpretará como cierto cuando la condición asociada a su primer argumento (que representa un evento) se cumpla. Esta condición procede de la declaración de relaciones de disparo realizada en la Especificación fuente.

Definición 4.29 *Dado un evento e(c,v,t), se define el metapredicado* **Trigger** *como:*

Trigger(e(c,v,t)) $\leftarrow$ *condición*

Esta condición es, como ocurría con la precondición del metapredicado Precond, *una fórmula de L convertida en una formula atómica de L' usando los metapredicados ya definidos.*

Si una condición de disparo se satisface, su evento asociado se activa. Ello permite completar la definición del metapredicado **insert** en entornos activos.

Definición 4.30 Animación activa de una especificación en Oasis. *Dada una historia h que representa el estado actual del sistema (secuencia de eventos relevantes), y un evento e se define el metapredicado* **insert** *como:*

* *Insert(h,e,[e | h]) ← Event(e),Precond(e).*

* *Insert(h,e,[e | h]) ← Trigger(e),Event(e), Precond(e).*

El uso de técnicas de metaprogramación posibilita la formalización de un entorno de especificación Oasis de una forma muy simple, que da cuenta también de la evolución dinámica o animación del programa lógico equivalente a una especificación.

Vamos a ver a continuación como también se puede abordar esta formalización desde un punto vista algebraico.

4.5.2 Eventos como operadores algebraicos

Hemos visto cómo se genera la presentación algebraica denotada por una especificación en Oasis. En este contexto formal algebraico, la presentación correspondiente a una especificación Oasis es la asociada a una signatura Σ, compuesta por:

1) un conjunto de géneros, cuyos miembros son los géneros correspondientes a las clases primitivas, elementales y compuestas declaradas en la especificación Oasis fuente.

2) un conjunto de operaciones compuesto por:

   * un conjunto de eventos como operaciones constructoras
   * un conjunto de atributos como operaciones consultoras, cuyos axiomas ecuacionales son los procedentes de la definición de dichos atributos en la especificación.

Tenemos pues todas las componentes necesarias para definir la noción de animación en este marco formal.

Definición 4.31 *Animar o Prototipar un sistema pasivo consiste en evaluar términos $t \in T_\Sigma$ en el Algebra de Términos $T_{\Sigma+PC+RI}$, donde:*

1) Σ es la signatura Σ asociada a una especificación en Oasis.

2) PC es el conjunto de operaciones booleanas que representan precondiciones asociadas a eventos:

$$PC_{evento_i} : Wff_{evento_i} \times Traza \rightarrow Boolean$$

3) RI es el conjunto de operaciones booleanas que representan restricciones de integridad estáticas definidas en la especificación Oasis fuente.

$$RI_i : Wff_i \times Traza \rightarrow Boolean$$

Este proceso de animación tiene dos vertientes:

* ejecución de observaciones, evaluando términos pertenecientes a la signatura Σ. Los eventos actúan como operaciones constructoras de términos (trazas), y los atributos son las operaciones consultoras cuyos axiomas proceden de la definición de los atributos en la especificación Oasis fuente.

  El estado de un objeto se representa a través de su traza. Dado un estado (una traza) observar el objeto es deducir el valor de sus atributos (operaciones consultoras) sobre dicha traza (compuesta por la secuencia de eventos que conforman la vida del objeto), que es un término $t \in T_\Sigma$.

* adición de eventos, o lo que es lo mismo, transición entre estados alcanzables. Los eventos actúan como constructores de instancias de una clase cambiando su estado. Esto solo ocurre si el estado en el que queda el objeto es un estado admisible. Un estado es admisible cuando se verifican las siguientes condiciones:

1) El estado pertenece al conjunto de estados válidos S_L, es decir, satisface las restricciones de integridad estáticas definidas para todo estado.

2) El estado es alcanzable, es decir no se viola la precondición que caracteriza la relación de accesibilidad.

Ambas condiciones se confirman evaluando Restricciones de Integridad estáticas en el término $t' \in T_\Sigma$ generado por el evento relevante considerado a partir del término t inicial, y Precondiciones de eventos en dicho término t. Sólo se producirá la transición al nuevo estado si unas y otros se evalúan a cierto.

El proceso de activación de eventos candidatos a ser insertados en la traza o ciclo de vida del Sistema puede realizarse de dos formas:

1) Eventos activados directamente por el entorno.

2) Eventos activados de forma espontánea por la actividad del propio Sistema.

La introducción de actividad en el sistema se realiza a través de las relaciones de disparo. En este contexto algebraico que hemos elaborado, las relaciones de disparo se representan como operadores booleanos que actúan como activadores de eventos cuando se evalúan a cierto. Esta evaluación, que se realiza de forma continua en paralelo con el proceso normal de animación anterior, constituye la actividad interna del Sistema.

Definición 4.32 *Las relaciones de disparo declaradas en una especificación Oasis dan origen al conjunto RD de operaciones de disparo:*

$$RD_{evento_i} : Wff_{evento_i} \times Traza \rightarrow Boolean$$

La actividad en el sistema considerado consiste en evaluar estas operaciones sobre los términos que representan el estado del sistema.

En el entorno de producción automática de Software que se presenta en este trabajo, un monitor activo es el encargado de testear satisfacción de condiciones de disparo y de generar el evento cuya operación de disparo

se evalúe a cierto, con la granularidad declarada en la especificación Oasis fuente.

En la figura 4.2 se muestra la arquitectura del entorno formal presentado. Partiendo de una especificación Oasis, un traductor de alto nivel genera la Teoría de Primer Orden correspondiente a dicha especificación. Esta Teoría tiene dos componentes:

1) El Algebra de Términos $T_{\Sigma+RI+PC}$ que representa el estado del Sistema y las observaciones que de él se pueden realizar.

2) El conjunto de operaciones de disparo RD que se intentan satisfacer continuamente para actuar como agentes de eventos, constituyendo la actividad interna del Sistema.

Los **Agentes** son activadores de eventos e inductores, por tanto, de cambios en los términos de $T_{\Sigma+RI+PC}$ que representan el estado de un sistema, si se cumplen las condiciones vistas anteriormente. Por último, los usuarios utilizan el Lenguaje de Primer Orden asociado a la Teoría como herramienta de elaboración de consultas y obtención de respuestas.

Presentado el modelo O^3 utilizado para describir la realidad, el entorno de especificación Oasis diseñado de acuerdo con él, y las propiedades formales que lo caracterizan, ha llegado el momento de presentar una metodología de producción de Software que recoja estas ideas y las englobe dentro de un Paradigma de Programación Automática basado en métodos formales, cubriendo las fases clásicas de Análisis, Diseño e Implementación. Esta nueva Metodología es **OO-Method**, y va a ser presentada a continuación.

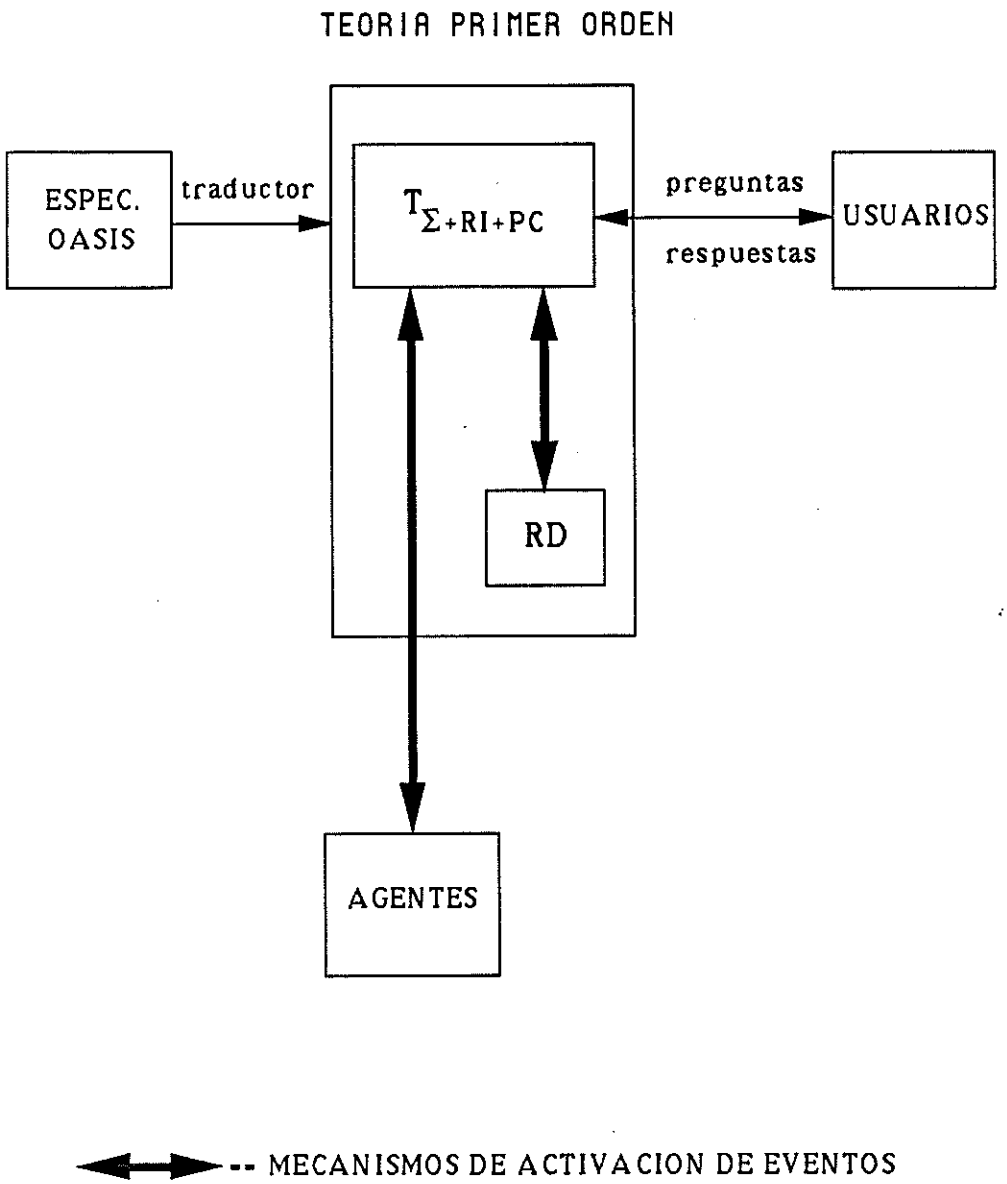


Figura 4.2: Arquitectura del entorno formal de prototipación y animación Oasis desde una perspectiva algebraica.

Capítulo 5

OO-Method:Una Metodología de Producción Automática de Software

5.1 Introducción

La experiencia adquirida en el uso industrial de tecnologías OO demuestra que la adopción de un proceso disciplinado en el desarrollo de Software es el factor esencial que determina el éxito o el fracaso de dicho desarrollo [30]. Las componentes fundamentales de tal proceso son técnicas de Análisis y Diseño sistemáticas, y en esa línea son varias las técnicas de Análisis OO (AOO) y Diseño OO (DOO) que han sido desarrolladas [15,28,82,98]. Todas esta técnicas tienen en común el uso de modelos semánticos tipo Entidad-Relación (E-R), máquinas de estados y diagramas de flujos de datos (DFD) para generar diferentes perspectivas del problema, como resultado de la fase de AOO.

El objetivo de todo AOO es facilitar la descripción y comunicación OO del Sistema que se quiere desarrollar. Obviamente, las notaciones empleadas deben conjugar expresividad y claridad para que su uso no suponga un nuevo problema añadido al problema del diseño del Sistema. Como se argumenta en [31], en un mundo imperfecto nada es perfecto, pero los diagramas E-R, las máquinas de estado y los DFD's se vienen usando con éxito para comunicar descripciones obtenidas durante el Análisis y el Diseño.

Si sólo el poder expresivo fuese un criterio suficiente para juzgar la notación de un Análisis, entonces con el Lenguaje Natural sería más que suficiente. Sin embargo, los modelos de un Análisis deben hacer posible la comunicación *precisa* acerca de dominios complejos. Ello hace que las notaciones empleadas deban ser:

* **Clara**, sin ambigüedades que puedan hacer que una visión incorrecta del problema sea analizada y desarrollada.

* **Abstracta**: los modelos del Análisis no deben contener aspectos no esenciales, o aspectos irrelevantes relacionados con la implementación.

* **Consistente**. Debe ser posible relacionar unos modelos con otros para evitar hipotéticas confusiones derivadas de descripciones inconsistentes.

El trabajo de todo analista está orientado a producir un conjunto coherente de modelos que constituyan una descripción completa del do-

minio del problema, usando una notación que tenga las propiedadades anteriores.

En esa línea, son muchas las metodologías propuestas, aunque los resultados no son siempre tan espectaculares como se quisiera. En [98] se propone el uso de tres tipos de modelos que expresan diferentes aspectos o vistas del Sistema:

* El modelo *funcional* expresa lo que pasa en el Sistema en términos de qué procesos son relevantes.

* El modelo *dinámico* permite expresar cuándo pasa: se preocupa del comportamiento temporal y usa máquinas de estados finitos extendidas para especificar el comportamiento de las clases.

* El modelo *de objetos* representa los aspectos estructurales del Sistema, basándose en un diagrama de tipo E-R para mostrar las clases componentes del Sistema y sus relaciones.

El método es totalmente informal, y las distintas notaciones utilizadas hacen difícil hablar con propiedad de consistencia. Los modelos resultantes del Análisis no constituyen un conjunto coherente de descripciones (efecto red) y la transición de Análisis a Diseño es prácticamente un proceso heurístico.

En [31] se mejora la Metodología anterior utilizando como modelo funcional una especificación declarativa basada en pre-post condiciones. Los modelos dinámico y funcional se expresan haciendo uso de los dominios de datos declarados en su *modelo de estructura de objetos*, lo que introduce un mayor grado de sistematización en el proceso, y hace posible hablar de consistencia, aunque la Metodología continúa siendo no formal.

En una línea similar pero introduciendo modelos OO, en [17] se presenta una Metodología que a partir de un Modelo Semántico extendido (*Morse*) permite obtener un Modelo Operacional OO (O^2).

En cualquier caso, el uso de diferentes notaciones con diferentes expresividades y distintas nociones, hace harto difícil el generar entornos de trabajo basados en métodos formales, y no eliminan el problema más clásico en cualquier entorno de producción de Software, relativo a la adecuación del producto final a los requerimientos iniciales.

El desarrollo de un entorno de trabajo declarativo abre por el contrario la puerta a nuevas soluciones que permitan abordar con otra mentalidad el proceso productivo de Software, más próxima a lo que es en realidad una ingeniería. Permite además hacer uso de todos los resultados teóricos y prácticos acumulados durante los últimos años en el terreno de la programación lógica.

El objetivo de este capítulo es desarrollar en esa línea una Metodología que cubra las fases típicas de un proceso de producción de Software de Análisis, Diseño e Implementación, recogiendo los principios anteriores dentro del contexto del modelo O^3, y consecuente con los principios del Paradigma de Programación Automática propuesto por Baltzer [7].

Esta Metodología es la aportación final del trabajo desarrollado, y proporciona un avanzado Entorno de Programación en el que se dan nuevas soluciones a los problemas clásicos denotados globalmente por el término 'Crisis del Software'. Concretamente:

1) Un primer problema de los entornos clásicos de producción de Software es el escaso esfuerzo dedicado a las fases de Análisis y Diseño.

   OO-Method aporta como solución la modificación del Ciclo de Vida Clásico introduciendo Prototipación Automática en un entorno declarativo y formal.

2) Otro problema clásico reside en el uso y abuso de Lenguajes Informales.

   El modelo O^3 en el que se sustenta OO-Method ha sido formalizado, y el Lenguaje de Especificación asociado al modelo, Oasis, es un lenguaje de especificación formal. Más aún, se ha mostrado como una Especificación escrita en Oasis es equivalente e una Teoría Formal de Primer Orden (clausal, ecuacional o clausal con igualdad) dependiendo de la versión utilizada.

3) Un tercer problema se deriva del uso de modelos inadecuados para describir la realidad, que en la mayoría de ocasiones fuerza a los diseñadores a combinar diferentes modelos, con distintas notaciones e inconsistencias entre ellos (modelos Entidad-Relación, Diagramas de Flujos de Datos,...).

El uso de un único modelo OO y Ontológico (O^3) proporciona un conjunto preciso de conceptos que hacen posible hablar de consistencia real entre los resultados de cada una de las fases del proceso de producción de Software.

4) La habitual complejidad de la interacción hombre-máquina en los entornos clásicos es reducida notablemente en OO-Method introduciendo avanzados entornos gráficos que permiten realizar una especificación gráfica del Sistema. Esta especificación gráfica es el resultado del Análisis, y su equivalente especificación en Oasis se obtiene automáticamente.

Vamos a ver en los siguientes apartados los métodos que conforman OO-Method, y que en resumen son:

1) El Análisis constituye la fase inicial del proceso de producción de Software y reune a usuarios y diseñadores de un Sistema en base a un vocabulario común procedente del dominio del problema. OO-Method proporciona una notación gráfica en un entorno amigable para generar dos especificaciones gráficas y OO:

   * Un modelo estructural del Sistema (**Diagrama de Configuración de Clases**).
   * Un modelo de comportamiento (**Diagramas de Estados de las Clases ('Classcharts')**), que incluye los aspectos dinámicos asociados a las clases especificadas en el modelo anterior.

2) Los modelos gráficos obtenidos como productos de la fase AOO se traducen de forma automática a una especificación en Oasis. Oasis es la herramienta de diseño de OO-Method.

3) La implementación se aborda de distinta forma dependiendo del tipo de entorno de programación utilizado.

   * En un entorno de Programación Lógica, como una especificación en Oasis es equivalente a una Teoría Formal de Primer Orden, la animación del programa lógico con el que se

representé dicha Teoría es el producto final, obtenido automáticamente a partir de la especificación en Oasis por medio de traductores de alto nivel.

En tal entorno se verifica la propiedad esencial que *el programa lógico producido es equivalente a la especificación fuente.*

* En un entorno clásico de programación OO, ya no tenemos las propiedades formales lógicas anteriores, pero de cualquier forma se produce automáticamente un prototipo del Sistema en el LPOO elegido.[1].

  No tenemos en este caso un producto final formalmente equivalente a la especificación, pero sí un conjunto de Programas que pueden ser generados de forma automática a partir de la especificación mediante un traductor, y que constituyen el punto de partida para la obtención del producto Software definitivo.

De esta forma se puede desarrollar una herramienta CASE OO que cubra todas las fases de desarrollo con un alto grado de automatización, y en un marco formal OO bien definido. Esta herramienta CASE, que hereda el nombre de la metodología, OO-Method, pone a disposición del usuario un conjunto de métodos operacionales que, a partir de una Especificación Gráfica del SO (AOO), genera su correspondiente Especificación en Oasis y en función del entorno de programación seleccionado, el producto Software final equivalente.

Para presentar la Metodología vamos a usar como ejemplo un SO consistente en un *hospital*, compuesto por doctores, pacientes y habitaciones individuales. El ingreso de un paciente en el hospital hace que éste ocupe una habitación disponible. Durante su estancia, uno o varios doctores están encargados de su atención médica. Estos doctores realizan servicios, que tienen una fecha y un precio asignado. Cuando se da de baja a un paciente, se prepara una factura que contiene todos los servicios prestados desde su ingreso.

[1]Las extensiones actuales de OO-Method en este contexto incluyen traductores para C++ y Smalltalk.

5.2 Análisis

Un AOO tiene siempre como objetivo la construcción de un modelo del mundo real a partir de una perspectiva OO del mundo. En [15] se define AOO como aquel método de Análisis que examina los requerimientos de un Sistema desde la perspectiva de las clases y objetos detectados en el vocabulario del dominio del problema. Smith and Tockey [120] lo definen como el proceso de identificación y modelización de las clases esenciales de objetos, y las relaciones lógicas e interacciones que existen entre ellos. Muchas más definiciones han sido propuestas, dando cuenta la mayoría de ellas de la misma noción con la misma ambiguedad.

En O^3, la noción de AOO está fundamentada en los conceptos básicos del modelo OO que se presentaron en el capítulo 2. Se define AOO desde un punto de vista doble:

1) Como un proceso de clasificación para identificar las clases existentes en el SO, y las relaciones existentes entre ellas. Para que la definición sea precisa hay que determinar qué tipos de relaciones se van a considerar. Veremos ahora mismo que estas relaciones son las de **agregación**, **herencia**,....

2) Para cada clase detectada se tiene que determinar cuáles son los estados posibles de sus instancias, y cuáles los eventos causantes de la transición de un estadó a otro, así como las hipotéticas condiciones asociadas a tales cambios de estado.

Como resultado del AOO se generan por tanto dos tipos de gráficos:

* Un **Diagrama de Configuración de Clases** (DCC). Es un Modelo Semántico Extendido que muestra la estructura estática del SO, pero que incluye también para cada clase sus eventos relevantes.

* Cada clase componente del DCC tiene asociado un **Diagrama de Estados de la Clase** (DEC) en el que se representan los estados potencialmente válidos para instancias de la clase, y los eventos causantes de transiciones entre dichos estados.

5.2.1 Diagrama de Configuración de Clases (DCC).

Introducción

Para poder representar la estructura de la Sociedad de Clases que componen el SO estudiado, hay que seguir dos pasos:

1) Un proceso de clasificación para determinar qué clases componen el SO.

2) Precisar la estructura de cada clase en términos de su conjunto de atributos y eventos.

El DCC es la herramienta gráfica que permite describir las clases componentes del SO, definiendo sus aspectos anatómicos y sus relaciones entre clases. De esta forma se obtiene una visión *estática* (arquitectónica) completa del SO como sociedad de objetos interactivos.

Vamos a ver en primer lugar cómo realizar ordenadamente un AOO en OO-Method, y a continuación se presentará una notación gráfica con la que representar los modelos gráficos resultantes.

Proceso de Análisis

El proceso de realización del AOO en OO-Method consta de cinco etapas:

1) **Identificación de clases**: se trata de identificar las clases elementales del dominio del SO.

   En nuestro ejemplo, consideramos cuatro clases elementales: doctores, pacientes, habitaciones y facturas.

2) **definición de atributos primitivos**. En esta fase se trata de precisar el conjunto de *atributos primitivos* de cada una de las clases detectadas en el punto 1. Un atributo primitivo es aquel cuyos tipos son TAD (Tipos Abstractos de Datos o Dominios). Hay que distinguir entre atributos constantes y variables.

   Volviendo al ejemplo, la clase *Doctor* tiene atributos constantes de tipo 'string' *numero* (de colegiado), *nombre*, *especialidad*, y un atributo variable de tipo 'integer' *num_servicios*, que indica el número de servicios prestados por un doctor.

3) **declaración de eventos.** De nuevo para cada clase del punto 1, se trata de declarar su conjunto de eventos relevantes, distinguiendo entre privados y compartidos, y entre los privados, el evento creador y destructor de instancias de la clase.

   Los eventos privados de la clase *Doctor* son *contratar, despedir*, que son los eventos generador y destructor de instancias respectivamente Existe asimismo un evento compartido *dar_servicio* entre las clases *Doctor,Paciente*, que relaciona un doctor con un paciente a través del servicio que ha sido realizado.

4) **relaciones entre clases.** Las clases de la Sociedad de Objetos sabemos que no son independientes. Existen relaciones jerárquicas estructurales entre ellas que vamos a modelar en términos de **agregación** (parte de) y **herencia** ('is-a')

   (a) La agregación es una relación estructural interclases, que posee una cardinalidad. Denota propiedades que relacionan dos o más clases. Cada propiedad tiene un nombre que la identifica unívocamente.

   La cardinalidad de una agregación es su propiedad fundamental, y puede ser de tres tipos:

   * de uno a uno (1:1)
   * de uno a muchos (1:M)
   * de muchos a muchos (M:N)

   En el ejemplo del *Hospital* tenemos una agregación entre las clases *Doctor, Paciente*, con una cardinalidad de M:N. El nombre dado a esta agregación es *Servicios*.

   (b) **Herencia.** Podemos definir brevemente la herencia como aquella relación entre clases en la que una clase hija comparte estructura y comportamiento definida en una (herencia simple) o muchas (herencia múltiple) otras clases (padre). Esta relación 'is-a', ampliamente usada en Modelización Semántica, constituye una potente herramienta semántica con la que manipular los conceptos de especialización y generalización.

   Como ejemplo, podría existir en nuestro *hospital* una nueva clase de *pacientes*, los *pacientes asegurados*, que serían aque-

llos pacientes adscritos a alguna compañía de seguros. Se trata de una clase hija de la clase *paciente*, que hereda las propiedades de su clase padre y añade las suyas propias como propiedades emergentes.

5) Una relación de agregación puede constituir una nueva clase compleja por sí misma. Esta situación se presenta cuando asociada a la relación existen propiedades emergentes, como nuevos atributos o eventos. En este caso, la relación original se convierte en una clase a la que debemos aplicar los puntos 1..4.

   Este caso es el de la relación *Servicios* definida como una agregación entre *Doctor,Paciente*. *Servicios* tiene como propiedades emergentes la fecha del servicio y su importe, que son atributos primitivos de la nueva clase.

Una notación gráfica para representar ordenadamente la información recogida en las fases anteriores permitirá obtener el DCC resultante del AOO, que será el modelo gráfico que da cuenta de los aspectos estáticos y estructurales del SO que se quiere diseñar.

Notación Gráfica

Las unidades básicas de la notación gráfica que se va a presentar son las clases y las relaciones entre clases.

Otros Lenguajes Gráficos de Especificación OO fijan su granularidad a nivel de objeto (instancia). En [9], por ejemplo,se utiliza un Diagrama de Configuración de Objetos (DCO) para representar instancias individuales. La notación gráfica de los DCC hereda algunas características de esos DCO, generalizados y enriquecidos al nivel de clase.

Evidentemente, un entorno de trabajo OO-Method está basado en clases, y el cómo representar gráficamente una clase será el primer aspecto a considerar.

Una vez propuesta una determinada representación, el mismo problema tendrá que ser resuelto al nivel de relaciones entre clases.

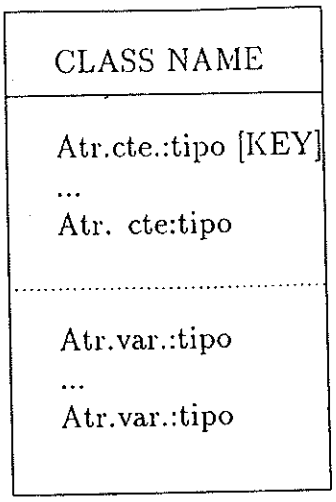


Figura 5.1: Representación gráfica de la estructura interna de una clase de un DCC en OO-Method

Representación de clases

En el DCC cada clase elemental se representa como una caja o rectángulo, en el que se incluye:

1) la declaración de sus atributos primitivos con su tipo. Se distingue entre atributos constantes y variables, declarándolos en ese orden. El o los atributos constantes que actúen como clave identificadora de objetos de la clase se marcan con la palabra reservada *key*. En la figura 5.1, se puede apreciar cuál es la representación gráfica de la estructura interna de una clase en un DCC.

2) la declaración de los eventos de la clase. La notación que vamos a usar tiene como antecesor la presentada en [9]. Allí todo objeto está provisto de un conjunto de servicios que pueden ser utilizados por otros objetos del Sistema, y otro conjunto de servicios que necesita le sean suministrados por otros objetos.

Los servicios del primer tipo se representan gráficamente por medio de líneas continuas, y los del segundo tipo por medio de líneas discontinuas. De esta forma, una conexión entre una línea continua y una línea discontinua muestra una posible comunicación entre los objetos del Sistema involucrados. Y una línea continua no conectada a ninguna otra denota una posible comunicación entre el objeto y el entorno del Sistema.

Esta notación es aprovechada para representar gráficamente eventos en un DCC. Ahora no se trata de asociar un conjunto de servicios a una clase, sino de asociarle un conjunto de atributos y eventos. Pero los eventos se pueden interpretar como servicios proporcionados por una (eventos propios) o más (eventos compartidos) clases, que modifican el estado de los objetos de la clase. Y los atributos también se pueden interpretar como servicios proporcionados por la clase, pero que no modifican su estado y solo se utilizan para efectuar observaciones y conocer el estado del objeto.

El concepto equivalente al de servicio requerido en los DCC es el de **evento activo**. Los agentes son activadores de eventos, por tanto actúan como peticionarios de eventos proporcionados por las clases del Sistema. Los agentes potenciales para cualquier evento dado son el entorno o cualquier objeto de la Sociedad de Objetos.

La inclusión de la información de qué agentes están asociados a un evento va a enriquecer la expresividad de la notación gráfica que se quiere desarrollar.

Se llega de esta forma a la conclusión de que el entorno gráfico de AOO, en la vertiente estructural representada por el DCC, debe proporcionar una notación que satisfaga los siguientes requerimientos:

* Diferenciar eventos propios y compartidos de una clase
* Precisar qué agentes van a poder actuar como activadores del evento

Cualquier evento de una clase se representa por medio de una línea continua que sale de la clase propietaria. Un evento será propio cuando esa línea continua no conecta con ninguna otra clase, y será compartido cuando sí exista tal conexión. En ese caso las clases conectadas serán las clases propietarias del evento compartido. Se satisface así el primer requerimiento.

Para todo evento de una clase, se indicarán cuáles son sus agentes por medio de líneas discontinuas, que enlazarán el evento con la clase cuyas instancias puedan actuar como agentes.

De esta forma, una línea discontinua que enlaze una clase A con una línea continua que represente un evento de otra clase B indica que instancias de la clase A pueden activar el evento en cuestión. Una línea discontinua que no empieza en ninguna clase indica que el entorno podrá actuar como agente. Situaciones mixtas se presentan cuando tanto el entorno como otras clases puedan actuar como agentes.

Por ejemplo, en la figura 5.2 se muestra un esquema de comunicación entre dos clases A y B, en el que la clase A tiene tres eventos propios, y un evento compartido con la clase B. Los eventos se representan por medio de líneas continuas, y sus agentes con líneas discontinuas según se ha establecido. En este caso, tenemos:

(a) una comunicación entre los dos objetos puede establecerse activando el evento que comparten, que tiene como agentes potenciales el entorno o instancias de cualquiera de las dos clases.

(b) una comunicación entre el entorno y objetos de la clase A puede establecerse a través de los eventos propios de la clase A, cuyo agente es el entorno.

En este momento, ya podemos representar gráficamente clases con su estructura interna, y con sus eventos relevantes distinguiendo los privados de los compartidos e identificando sus potenciales agentes. Un agente puede activar un evento de forma espontánea, o como consecuencia de la satisfacción de una condición de disparo. Abordemos ahora la

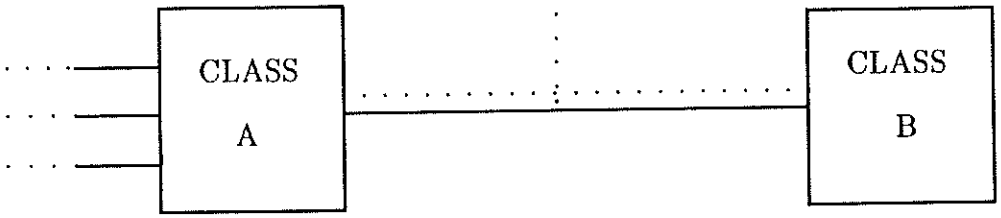


Figura 5.2: Representación gráfica de comunicación entre clases en un DCC

representación gráfica de las relaciones entre clases.

Relaciones entre clases

La notación gráfica que se está elaborando debe permitir especificar los dos tipos de relaciones estructurales entre clases relevantes en OO-Method: agregación y herencia.

Agregación

Se utiliza para indicar que un objeto se compone de otro(s) objeto(s). Permiten expresar relaciones entre clases caracterizadas por su cardinalidad (dado un objeto, el número de instancias del otro objeto con el que se relaciona y viceversa). Cada relación de agregación tiene un nombre que la identifica unívocamente. La gestión de estas relaciones en tiempo de diseño veremos que constituye un aspecto fundamental en la transición de Análisis a Diseño en OO-Method.

Gráficamente, los distintos tipos de agregación se representan como se indica a continuación:

1) Una agregación de cardinalidad 1:1 se representa por medio de una línea gruesa[2] que conecta las clases relacionadas, con un pequeño círculo opcional en los extremos. Si el círculo está presente,

[2]Por línea gruesa denotaremos de ahora en adelante una línea de espesor sustancialmente mayor que el de las líneas continuas con las que se representan los eventos

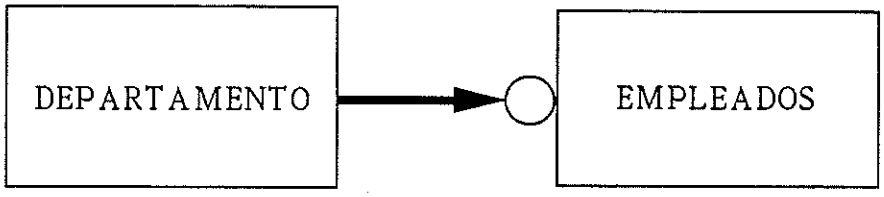


Figura 5.3: Ejemplo de agregación de cardinalidad 1:M, con pertenencia opcional para la clase apuntada por la flecha

queño círculo opcional en los extremos. Si el círculo está presente, la pertenencia a la agregación de la clase a la que el círculo está pegado es opcional. En caso contrario, es obligatoria.

2) Una agregación de cardinalidad 1:M se declara gráficamente conectando las clases relacionadas con una flecha gruesa, cuya cabeza apunta a la clase cuya cardinalidad es de muchos. Nuevamente, la adición de un pequeño círculo en cualquiera de los extremos de la linea indica que la pertenencia de la clase correspondiente a la agregación es opcional, admitiendo cardinalidades 0.

   El ejemplo archiconocido del *departamento* y sus *empleados*, relacionados por una agregación de 1:M sirve para ilustrar lo anterior. Si un departamento puede no tener empleados, entonces la pertenencia de la clase *empleado* a la agregación es opcional (o lo que es lo mismo, M puede denotar 0,1 o muchos empleados). Gráficamente la representación sería la que se muestra en la figura 5.3. Tal y como se declara esta agregación, un empleado debe estar obligatoriamente relacionado con un departamento.

3) Una agregación de muchos a muchos (M:N) se representa mediante una flecha constituida por una linea continua gruesa, con dos cabezas que apuntan a las clases participantes.

Como comentamos en el apartado anterior, una agregación puede constituir una nueva clase cuando tiene propiedades emergentes. En

este caso siempre existe un evento compartido entre las clases relacionadas que actúa como soporte de la agregación, y que es el evento creador de instancias de la nueva clase. En la notación gráfica de los DCC, esta situación se representa a través de una nueva caja dibujada sobre la agregación que le da origen.

En la figura 5.4 se puede ver el DCC correspondiente al ejemplo del *hospital*. En él se omite la declaración de atributos primitivos. En el entorno gráfico implementado, tal información es siempre accesible seleccionando con el ratón una clase y presionando el botón correspondiente. Vemos c omo la agregación *servicios* constituye una nueva clase, representada por medio de una nueva caja sobre la flecha de la relación. La agregación *localización*, que relaciona un paciente con la habitación que ocupa, no tiene entidad de clase, ya que no aporta propiedad emergente alguna.

Pueden representarse relaciones de agregación entre más de dos clases. La notación gráfica en esa situación es la misma. Para establecer la cardinalidad de cada clase participante en la agregación, se determina la cardinalidad con que participaría esa clase en una relación binaria entre la clase en cuestión y el conjunto restante de clases de la agregación.

Herencia

El segundo mecanismo de dependencia estructural interclases es la herencia. Una clase puede heredar estructura (atributos) y comportamiento (eventos, precondiciones, relaciones de disparo) de otras clases del Sistema. Consideramos la noción de herencia utilizada en Modelización Semántica a través de las Relaciones 'Is-a'.

La representación gráfica se efectúa a través de un triángulo que conecta la clase padre con su(s) clase(s) hija(s), como se muestra en la figura 5.5. La herencia múltiple se representa conectando una clase hija con más de una clase padre, como se muestra en la figura anterior.

Se ha caracterizado ya la notación gráfica propia de los DCC, con los que podemos representar la estructura estática del Sistema que se diseña. Es el momento de estudiar qué modelos gráficos acompanarán al DCC para dar cuenta de los aspectos dinámicos. Estos modelos gráficos van a ser los **Diagramas de Estados de las Clases ('Classcharts')**.

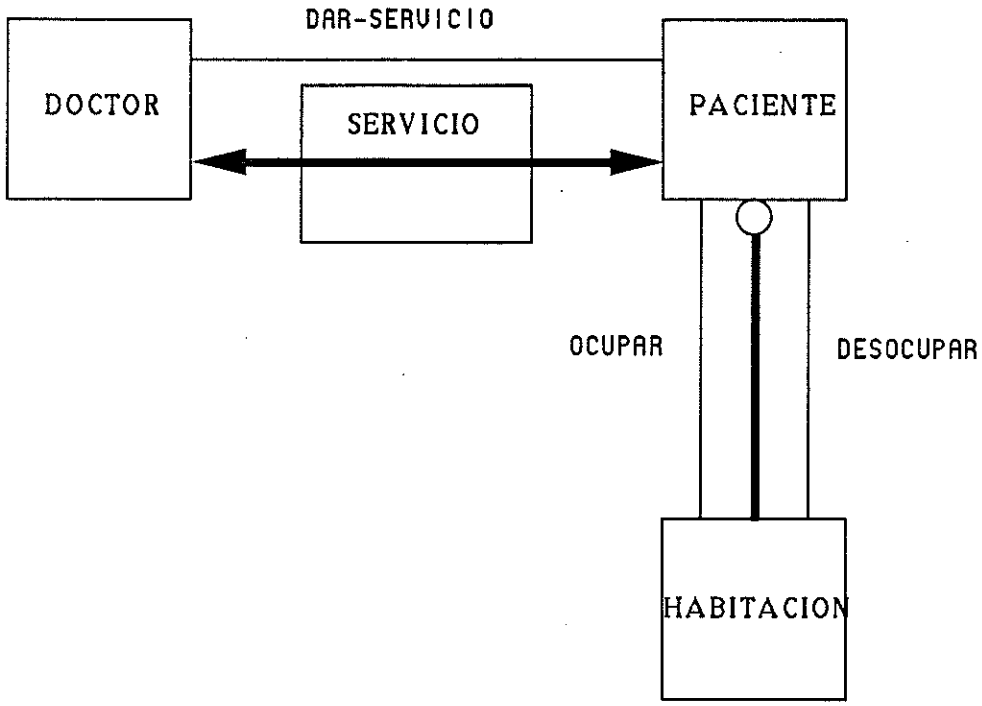


Figura 5.4: DCC correspondiente al ejemplo del Hospital

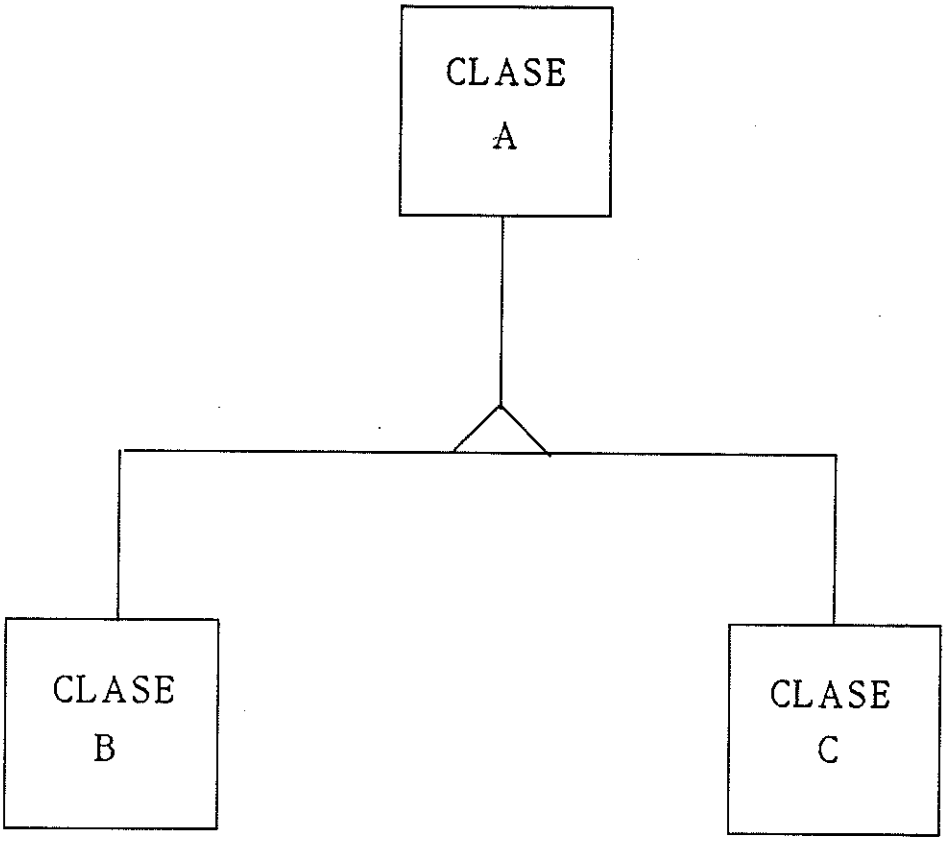


Figura 5.5: Representación gráfica de herencia en un DCC

5.2.2 Diagramas de Estados de las Clases (DEC)

La notación gráfica utilizada en los *DECs* es una extensión de la propuesta *Objectchart* [9,32], en la que un Objectchart es una máquina de estados finitos presentada en forma de Diagrama de Estados de Harel [59]. El efecto de los eventos sobre los atributos se define a través de la especificación de pre-post condiciones.

Partiendo de esta idea, vamos a definir un **DEC** para cada clase especificada en el DCC, en el que se incluirá la información de:

* estados potenciales en que pueden encontrarse los objetos de la clase.

* transiciones entre estados, etiquetadas por el nombre del evento causante del cambio de estado.

* precondiciones asociadas a la ocurrencia de un evento.

* posibles disparos resultantes de una transición de estado (ocurrencia de un evento)

Las etapas a seguir para sistemizar la obtención de toda la información anterior son las siguientes. Para toda clase se tiene que:

1) determinar el conjunto de estados potencialmente válidos. Cada estado tiene asociado un conjunto de atributos que lo caracterizan. Los objetos de una clase sólo pueden alcanzar o abandonar estados a través de la ocurrencia de eventos. Cada transición de estado está etiquetada con el nombre del evento que la produce.

2) describir el comportamiento del objeto a través de la definición de las trazas correctas. Los eventos no pueden ocurrir aleatoriamente. Caracterizar trazas válidas significa determinar cuando un evento se considera relevante y se añade a la traza que representa la vida del objeto(s) afectado(s).

3) determinar la dependencia del estado del objeto (conocido a través del valor de sus atributos) con respecto a sus eventos relevantes, con el objetivo último de poder diseñar automáticamente la función de observación en la fase de Diseño.

La propuesta Objectchart enriquece los Diagramas de Estados de Harel añadiendo información sobre atributos. Con los DECs vamos más allá, categorizando atributos e incluyendo esa nueva información en el modelo gráfico.

Descripción del comportamiento de los objetos de una clase

Sabemos ya que los objetos de una clase tienen un ciclo de vida, y cambian de estado como consecuencia de la ocurrencia de eventos. El comportamiento de los objetos de una clase puede por tanto representarse por medio de un Diagrama de Estados en el que se precisen las distintas etapas por las que pueden pasar, indicando cómo se produce ese 'paso'. Esas distintas etapas son los estados de la clase, y esos mecanismos de paso, los eventos.

Empezamos por tanto representando los estados válidos utilizando una notación gráfica basada en la composición AND de Harel, para poder expresar simultaneidad de estados. Cada estado se identifica unívocamente con un nombre. Los eventos se representan como arcos entre estados (transiciones de estado) etiquetados con el nombre del evento.

En la figura 5.6 se presenta el esqueleto de un DEC de acuerdo con la notación introducida. A continuación hay que especificar las posibles precondiciones asociadas a cada evento. La representación gráfica que se está desarrollando es especialmente apropiada para dar cuenta de ellas, porque asociado a todo evento se tiene una precondición implícita que establece que tal evento será relevante sólo si el objeto está en el estado adecuado. Un evento que origina un cambio desde un estado x a otro estado y sólo ocurrirá si el objeto considerado cumple la precondición de estar en el estado x.

Cualquier otra precondición que deba satisfacerse para que un cambio de estado se produzca, se declara en el arco que representa al evento como fórmulas de primer orden construidas sobre átomos constituidos por eventos u operadores relacionales cuyos operandos sean atributos y valores de atributos. Se trata de las precondiciones definidas en el capítulo 2. La sintaxis utilizada es:

evento [if precondición]

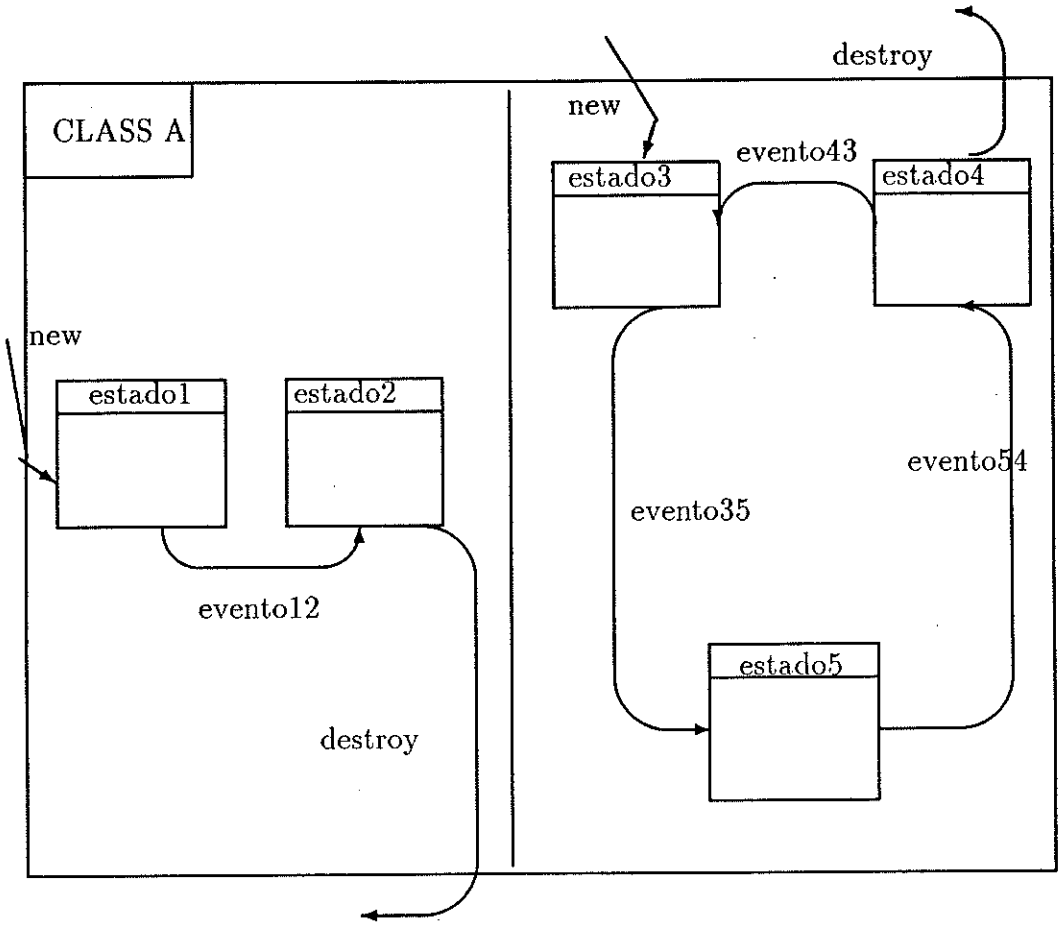


Figura 5.6: Estructura básica de un DEC. En ella se representa como los objetos de la clase A pueden estar en el estado1 o el estado2, y en el estado 3 o el estado 4 o el estado5, etiquetando cada transición con el nombre del evento que la activa. Los eventos new/destroy indican respectivamente los estados en los queda un objeto cuando es creado/los estados en que debe estar para poder ser destruido.

Un evento puede ser activado como consecuencia de la ocurrencia de otro evento (relaciones de disparo). Esta situación se representa gráficamente en un DEC como:

evento [if precondición] [/evento_disparado [if precond_disparo]

donde el evento *evento_disparado* se activa cuando el evento *evento* es relevante (es decir, su precondición se evalua a cierto) y se satisface la condición (opcional) de disparo *precond_disparo*.

En conjunto, los DECs extienden la visión estática del Sistema producida por el DCC. Se tiene una notación gráfica con la que representar los dos mecanismos de interacción entre objetos considerados en OO-Method: compartición de eventos y relaciones de disparo, proporcionando una herramienta expresiva capaz de especificar mecanismos activos de comunicación interobjetuales.

Para terminar, en la figura 5.7 se presenta el DEC correspondiente a la clase *habitación* del ejemplo. Los objetos de la clase *habitación* pueden estar en dos estados: 'ocupada' o 'libre'. Las transiciones de un estado a otro las producen los eventos ocupar y desocupar. El estado 'libre' tiene un atributo *disponible*, que es de categoría 'bool'. Veremos a continuación la noción de *categoría* de un atributo.

Observabilidad de los objetos

Para completar la información contenida en un DEC, hay que precisar cuáles son los atributos que caracterizan un estado dado (*atributos de estado*), determinando cómo cambian sus valores en función de sus eventos relevantes. Veremos al abordar la fase de Diseño cómo esa información permite especificar automáticamente la definición de los atributos variables, lo que implica dar cuenta de la observabilidad de los objetos de una clase a partir de los modelos gráficos resultantes del Análisis de forma automática.

Dos preguntas deben ser contestadas para obtener esa información adicional:

1) ¿Qué quiere decir que un atributo es relevante para un cierto estado?

2) ¿Qué atributos son relevantes a qué estados?

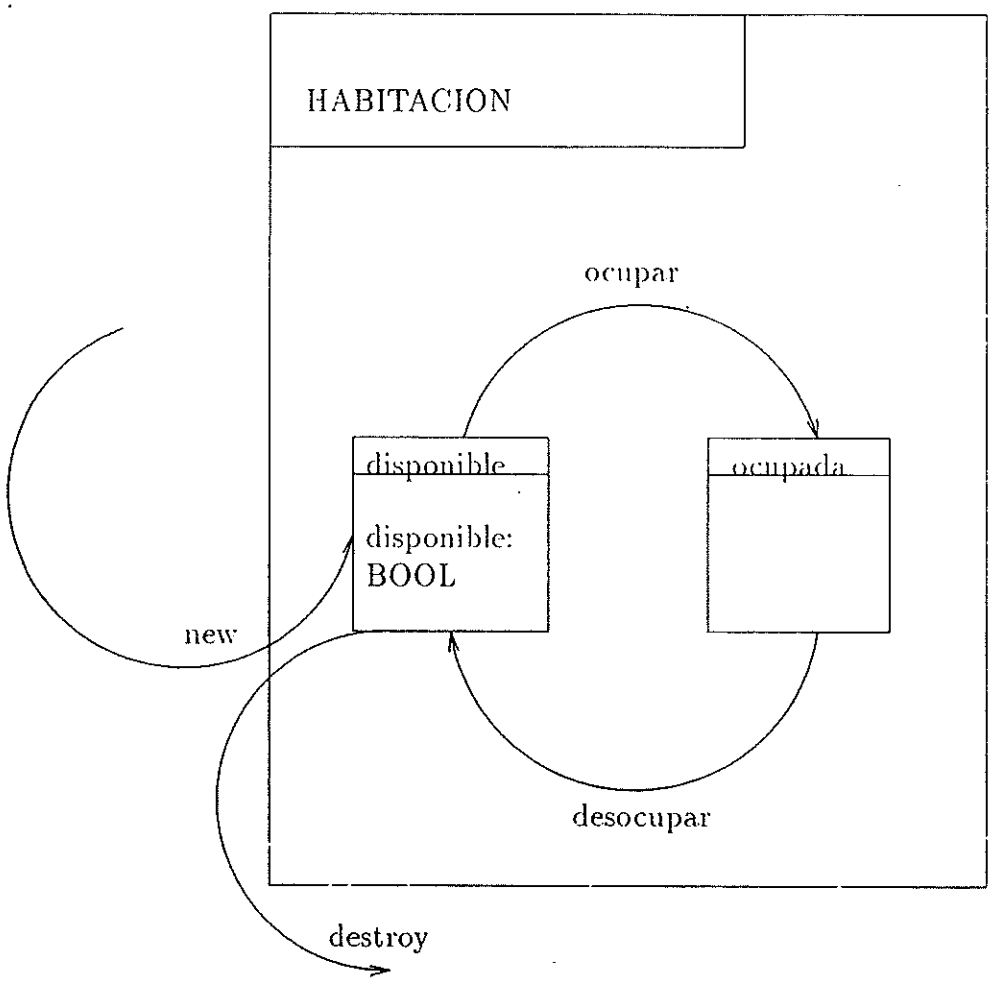


Figura 5.7: DEC correspondiente a la clase *habitación*

Dado un estado e, diremos que un atributo a es **relevante** para e cuando los únicos eventos que modifican los valores de a son los que llevan al objeto al estado e, o los que le hacen abandonarlo.

Todo DEC asociado a una clase incluye el conjunto de atributos relevantes para esa clase.

Aquellos atributos que tengan la propiedad de tener asociada una cardinalidad dinámica (es decir, un valor que aumenta y/o disminuye durante la vida del objeto dependiendo de ciertos eventos), se van a considerar atributos relevantes de un estado cuyos eventos asociados van a ser los que modifican la cardinalidad del atributo, ya sea de forma incremental o decremental. A este estado se llega siempre a partir de un estado inicial en el que el valor del atributo tiene cardinalidad nula. Esta aportación de la notación gráfica de los DEC elimina el problema de los potencialmente infinitos estados en que se puede encontrar un objeto, si para cada valor de tales atributos se tuviese un estado diferente.

Un ejemplo de esta situación se tiene con el atributo *número de servicios* de la clase *doctor*. Se trata de un atributo de tipo entero, cuyo valor (cardinalidad) es dinámico, siendo el evento *dar_servicio* el que lo modifica de forma incremental. Según lo expuesto en el párrafo anterior, tendremos un estado inicial (sin servicios prestados) y un estado final (con servicios prestados), con el atributo *número de servicios* como atributo relevante, como se indica en la figura 5.8. Para fijar de qué forma los atributos varían en función de los eventos relevantes del estado en que se encuentran, vamos a categorizar los distintos tipos de atributos que se pueden tener dependiendo de cómo sus eventos relevantes interaccionan con ellos. La adición al DEC de esta información adicional nos va a permitir diseñar la función de observabilidad asociada a toda clase.

Las categorías de atributos que vamos a distinguir son las cuatro siguientes:

* Atributos de **pertenencia a un estado** (denotados como **bool**): se trata de atributos que proporcionan información sobre si un objeto dado se encuentra en el estado considerado. Dicho estado es alcanzable a través de la ocurrencia de un evento el, llamado *evento portador*, y es abandonado tras la ocurrencia de otro evento

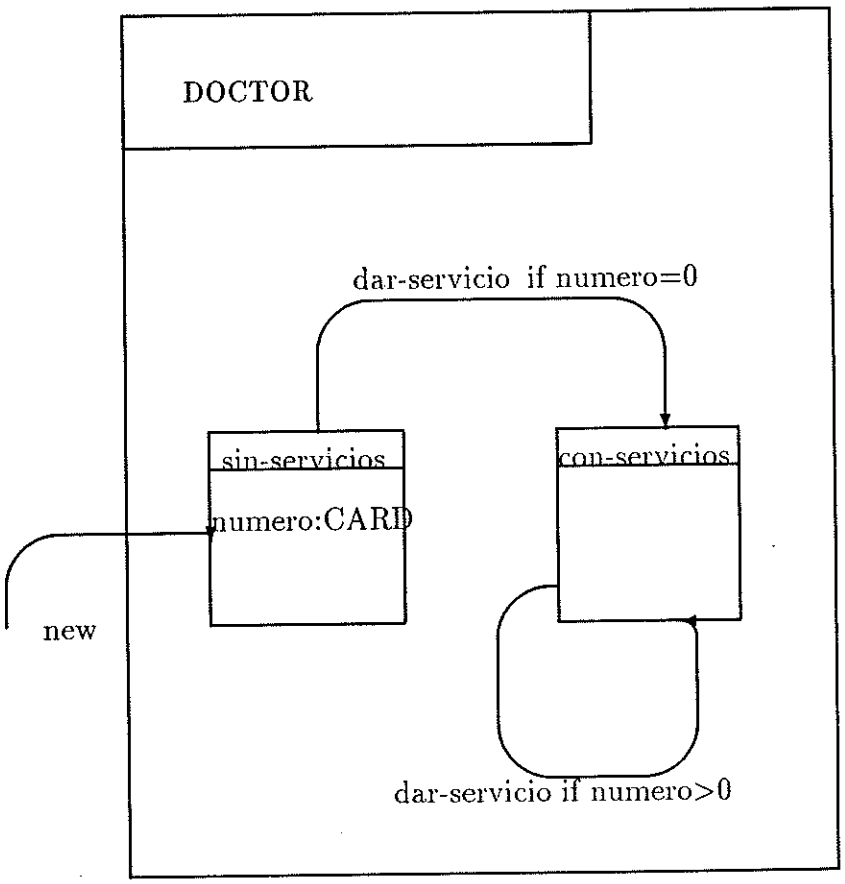


Figura 5.8: Representación de atributos con cardinalidad en un DEC. El atributo *número de servicios* de la clase *doctor* tiene asociada una cardinalidad, y se representa como atributo relevante de los estados inicial y final doctor *sin servicios* y *doctor con servicios* respectivamente.

$e2$, llamado *evento liberador*.

El conocimiento de los eventos que actúan como portador y liberador nos permitirá diseñar la definición del atributo.

Como ejemplo tenemos el atributo *disponible* de la clase *habitación*. Si una instancia de la clase está en el estado *ocupada* el valor del atributo es *true*, y si no *false*. Por tanto, *disponible* es un atributo de pertenecia al estado *ocupada*, cuyo evento portador es *ocupar*, y cuyo evento liberador es *desocupar*.

* Atributos **con cardinalidad** (denotados como **card**): son atributos caracterizados por tener asociado un evento $e1$ que incrementa su valor, y un evento $e2$ que lo decrementa. Las operaciones asociadas a esos eventos dependen del tipo del atributo (suma/resta si es entero, real etc., inserción/borrado si se trata de colecciones,..).

  Como se expresa en la figura 5.8, en este caso se especifica un estado inicial correspondiente a la situación de cardinalidad nula (inicial), y un estado final en el que los eventos $e1$ y $e2$ van modificando la cardinalidad (el valor) del atributo.

  Nuevamente, fijado cuál es el evento incrementador y el decrementador asociado a los estados inicial y final, podremos elaborar la definición del atributo en la fase de diseño.

  El atributo *número de servicios* de la clase *doctor* es un atributo con cardinalidad, con evento incrementador *dar servicio*, y sin evento decrementador.

* Atributos **característicos de un estado** (denotados por **state**). Estos atributos tienen un valor asociado que les es dado por el evento que deja al objeto en el estado considerado (evento portador). Se llaman característicos de un estado porque el valor que tienen es una propiedad del estado alcanzado por el objeto. Este valor es un parámetro del evento portador.

  La información que va a permitir diseñar la definición del atributo va a ser el evento relevante, con el parámetro que da valor al atributo.

El evento *ocupar* de la clase *paciente* lleva a un objeto *paciente* al estado *ingresado*. Un atributo característico de este estado es el *número de habitación*, un parámetro de tipo entero del evento *ocupar*. Una ocurrencia correcta de este evento lleva a un paciente al estado *ingresado*, dando valor a su atributo *número de habitación* asociado.

* Finalmente, se introduce una última categoría, que es la de los atributos **derivados**, definidos a partir de otros atributos pertenecientes a las categorías anteriores.

  Podríamos estar por ejemplo interesados en definir un atributo de tipo booleano a de la siguiente forma: si b es un atributo con cardinalidad (de tipo entero), y c es un atributo de pertenencia a un estado,

  $$a = true\ if\ a \geq 6\ and\ c = false\ otherwise false$$

  El estado para el que un atributo derivado es relevante no tiene explícitamente eventos portadores ni liberadores[3], ya que se trata de un caso claro de derivación de datos. El atributo lleva asociada su definición en función de otros atributos a través de una fórmula de primer orden.

  Esta definición va a permitir definir directamente el atributo en cuestión en la fase de diseño.

Caracterizados los modelos obtenidos como resultados de la fase de Análisis (el DCC, y un DEC por cada clase del DCC), vamos a ver ahora de qué forma es utilizada esa información para obtener una especificación en Oasis, que es la herramienta de diseño propuesta por OO-Method.

Como veremos, la conversión de los modelos gráficos resultantes en una especificación en Oasis es un proceso sistemático y mecanizable a través de un programa interactivo que cuando se presentan diversas alternativas de Diseño, las presenta en pantalla y espera la respuesta o decisión del Diseñador.

[3]Aunque sí los tiene de forma implícita, a través de los eventos relevantes para los atributos usados en su definición.

5.3 Diseño

Una de las principales ventajas derivadas del uso de un modelo OO es la natural evolución de los modelos resultantes del Análisis a lo que va constituir el resultado del Diseño. OO-Method es absolutamente fiel a tal noción, proporcionando una evolución natural en la que los conceptos manipulados son similares y en la que el modelo subyacente es siempre la propuesta OO O^3.

Tal aproximación permite a los Ingenieros de Software comunicar resultados de las dos fases de una forma precisa y en un entorno bien definido, a un nivel de abstracción totalmento ajeno a consideraciones de bajo nivel (implementación).

Las etapas que componen la fase de Diseño en OO-Method son las siguientes:

1) definición de clases, y para cada clase, declaración de eventos y atributos a partir del DCC.

   En esta fase, diferentes decisiones de diseño se presentan al manipular las relaciones entre clases del DCC.

2) a partir del DEC asociado a cada clase, completar su definición incluyendo:

   (a) definición de atributos variables, a partir de la información asociada a los atributos.

   (b) declaración de precondiciones y relaciones de disparo, a partir de la información asociada a las transiciones entre estados (eventos)

Vamos a ver con detalle cada uno de estos pasos, obteniendo como resultado final del diseño una especificación en Oasis.

5.3.1 Tratamiento de la información incluida en el DCC

Como resultado del Análisis en el DCC tenemos:

* clases elementales componentes del SO, incluyendo atributos primitivos y eventos propios y compartidos.

* relaciones entre clases (de agregación y de herencia) etiquetadas con un identificador.

Cada clase elemental presente en el DCC se convierte en una clase elemental de la especificación Oasis equivalente, en la que se declaran

1) los atributos constantes y variables cuyo tipo es un dominio (atributos primitivos)

2) los eventos propios y compartidos

a partir de la información asociada a cada clase en el DCC.

Relaciones entre clases

Herencia

Las relaciones de herencia se pueden representar en Oasis a través del operador de especialización o generalización. La elección de una u otra opción es una decisión de diseño. Una relación de herencia en el DCC entre una clase padre A y dos clases descendientes X e Y, puede dar origen en Oasis a dos declaraciones distintas:

1) Dos clases especializadas:

complex class X especialization of A
...
end complex class

complex class Y especialization of A
...
end complex class

2) Una clase generalizada:

complex class A generalization of X,Y.

En Oasis, la independencia de identificación se detecta automáticamente cuando la clave de una clase especializada es distinta de la clave de la clase generalizada correspondiente (o viceversa). En interpretaciones menos restrictivas de la herencia se trata de una decisión de diseño, para permitir situaciones en las que clase padre e hija tengan el mismo atributo clave, pero sus vidas sean independientes.

Agregación

El tratamiento de la información relativa a relaciones de agregación presenta una casuística mayor. En función de la cardinalidad asociada a la relación, se tienen diferentes opciones de Diseño:

* Una relación de cardinalidad 1:1 sin propiedades emergentes se transforma en un atributo objeto-valuado cuyo nombre es el de la relación. Este atributo puede declararse en cualquiera de las clase participantes en la relación, o en las dos, dependiendo del criterio del diseñador.

  Si la relación considerada tiene propiedades emergentes, entonces se transforma además en una clase, caracterizada por dichas propiedades emergentes más dos atributos constantes que referencien las instancias de las clases relacionadas.

* Una relación de cardinalidad 1:M tiene dos posibles representaciones en Oasis:

  1) Transformar la clase apuntada por el 'uno' de la relación en una clase compleja asociada de Oasis, enriqueciendo la información predefinida en dicha clase con los mecanismos de agrupación propios del operador asociación.

     Opcionalmente, el diseñador puede introducir un atributo evaluado a objeto en la clase apuntada por el 'muchos', para etiquetar cada instancia de esa clase con su instancia de clase asociada correspondiente.

  2) Si la agregación posee propiedades emergentes, y en el DCC se le ha dado categoría de clase, se declara tal clase como clase compleja agregada de la especificación Oasis.

* Con una relación de cardinalidad M:N tenemos de nuevo dos posibilidades:

  1) descomponer la relación de M:N en dos agregaciones de 1:M, definiendo las dos clases agregadas como clases complejas por asociación (mútuamente).

  2) si la relación de agregación tiene sus propios atributos y/o eventos, se declara como clase compleja agregada.

5.3.2 Conclusión de la especificación en Oasis usando los DEC's

Hasta ahora se dispone de un patrón de la especificación que está incompleto porque le falta la información asociada al comportamiento del Sistema. Esta información la componen en el entorno de OO-Method:

* la definición de los atributos variables

* declaración de precondiciones asociadas a los eventos

* declaración de relaciones de disparo, que introducen actividad en el SO.

Definición de atributos variables

Dada una clase, su correspondiente DEC proporciona el conjunto de atributos asociados a los posibles estados de los objetos de la clase. Cada atributo está etiquetado con su categoría (de pertenencia a un estado, con cardinalidad, característicos de un estado o derivados).

Conociendo la categoría del atributo y sus eventos relevantes, vamos a ver cómo se dispone de información suficiente para dar la definición del atributo en cualquiera de las versiones de Oasis.

Atributos de pertenencia a un estado

Dado un atributo de pertenencia a un estado, de tipo booleano, se necesita conocer cuáles son los eventos portador y liberador (que llevan a los

objetos de la clase considerada al estado e, o hacen que lo abandonen respectivamente).

Sea *a* un atributo de ese tipo, perteneciente a una clase C, que nos indica si un objeto dado está en el estado para el que *a* es relevante. Si este estado se alcanza tras la ocurrencia de un evento portador *ep*, y se abandona cuando ocurre un evento liberador *el*, la definición del atributo *a* es:

```
a(C,b,t)  if (ep(C,b,t) and b=true) or
             (el(C,b,t) and b=false) or
             (t1 is t-1 and a(C,b,t1))
```

en la versión clausal de Oasis,

```
a(C)=true if T=T1'ep(C,t)
a(C)=false if T=T1'el(C,t)
     else a(C) at T1
```

en la versión funcional, o

```
a(C,t)=true if ep(C,t)
a(C,t)=false if el(C,t)
     else a(C,t)=a(C,t-1)
```

en la versión clausal con igualdad.

Un ejemplo de tal atributo vimos que era el atributo *disponible* de la clase *habitación*. Clasificado como atributo de pertenencia a un estado, sabiendo que es atributo relevante del estado *ocupada*, y que su evento portador es *ocupar* y su evento liberador es *desocupar*, podemos escribir su definición en C-Oasis[4] como:

```
disponible(habitacion,bool,time) (false)
  clauses p:paciente;h:habitacion;b:bool;t,t1:time;
    disponible(h,b,t) if (ocupar(p,h,t) and b=false) or
                         (desocupar(p,h,t) and b=true) or
                         t is t-1,disponible(h,b,t1).
```

[4]La definición en F-Oasis y L-Oasis es similar y se deduce fácilmente del patrón anterior. En los siguientes ejemplos se da la definición práctica en C-Oasis, aunque previamente se muestra la estructura teórica haciendo uso de las tres expresividades.

Atributos con cardinalidad

Cuando un atributo es de esta categoría, en el caso más general tendremos dos eventos: uno que incrementa su cardinalidad en una cantidad n indicada por un parámetro del evento, y otro que la decrementa de forma análoga.

Dependiendo de la clase del atributo, las operaciones asociadas a tales incrementos y decrementos serán unas u otras. Por ejemplo, con atributos de clase integer, real, etc. incrementar significa sumar y decrementar restar. En atributos de clase colección (conjuntos, listas, pilas,..), incrementar significa insertar una nueva componente, decrementar implica eliminar una componente.

La definición de un atributo tal, se puede obtener a partir de la identificación de sus eventos asociados. Sea a un atributo con cardinalidad de una cierta clase C, y sean e+ el evento que aumenta la cardinalidad del atributo n' unidades, y e- el evento que la reduce n" unidades. Entonces escribiremos para C-Oasis:

```
a(C,n,t) if (t1 is t-1 and a(C,n1,t1) and
             (e+(C,n',t) and n=n1+n') or
             (e-(C,n",t) and n=n1-n")).
```

Para F-Oasis escribiremos:

```
a(C)=n if T=T1'e+(C,n',t) and a(C)=n-n' at T1.
a(C)=n if T=T1'e-(C,n",t) and a(C)=n+n" at T1.
             else a(C) at T1.
```

Y, finalmente, para L-Oasis tendremos:

```
a(C,t)=a(C,t-1)+n' if e+(C,n',t)
a(C,t)=a(C,t-1)-n' if e-(C,n",t)
          else a(C,t)=a(C,t-1)
```

Como ejemplc de atributo con cardinalidad estaba el *número de servicios* de la clase *doctor*, cuyo evento incrementador era *dar_servicio*, y que no tenía evento decrementador.

Su definición en C-Oasis es la siguiente:

```
number(doctor,int,time)
  clauses d:doctor;p:paciente;i,i':int;t,t1:time;
   number(d,i,t) if ((t1 is t-1 and number(d,i',t1)) and
                    (dar_servicio(d,p,t) and i=i'+1)).
```

Si el evento incrementador o decrementador considerado no tiene un parámetro que indique la cantidad que hay que añadir o eliminar al valor del atributo, se asume que tal cantidad es 1.

Atributos característicos de un estado

Si el atributo considerado es de la categoría *característico de un estado*, conocido el evento que hace ese estado alcanzable dando un valor al atributo, podemos escribir fácilmente su definición.

Sea *ec* tal evento, a el atributo en cuestión y C la clase a la que pertenece. Tenemos que en C-Oasis:

```
a(C,v,t) if (ec(C,v',t) and v is v') or
            (t1 is t-1 and a(C,v,t1)).
```

En F-Oasis:

```
a(C)=v  if T=T1'ec(C,v',t) and v=v'.
        else a(C) at T1.
```

Y en L-Oasis:

```
a(C,t)=v if ec(C,v,t)
   else a(C,t)=a(C,t-1)
```

El ejemplo que se daba al definir esta categoría de atributo era el del *número de habitación (#hab)* de la clase *paciente*. El evento *ocupar* lleva a un paciente al estado *ingresado*, del que el *#hab* es atributo característico. Identificado el evento característico y su parámetro relevante para el atributo, podemos escribir su definición en C-Oasis como:

```
#hab(paciente,habitacion,time) (0)
  clauses p:paciente;h,h':habitacion;t,t1:time;
     #hab(p,h,t) if (ocupar(p,h',t) and h=h') or
                    (t1 is t-1,#hab(p,h,t1)).
```

Atributos derivados

La definición de un atributo derivado es muy sencilla, ya que es una transcripción de la fórmula de primer orden con la que se le ha caracterizado en el DEC correspondiente. Esa fórmula da siempre el valor del atributo considerado en función de otros atributos de las categorías anteriores, definidos previamente.

Declaración de Precondiciones

Hemos visto como el DEC asociado a una clase contiene información sobre las transiciones de estado correspondientes a las ocurrencias de eventos. Cada transición está etiquetada por el evento que la produce. Esta representación introduce una primera componente para la precondición asociada a todo evento: para que una transición de estado se produzca, el objeto debe estar en un estado inicial de dicha transición. Siempre se puede dar cuenta de esta situación a través de un atributo de pertenencia a un estado, cuyo valor ha de ser *true* para que la ocurrencia de evento se produzca.

Componentes adicionales de la precondición asociada a un evento aparecen explícitamente en el DEC, en la información asociada al arco que representa al evento. Recordemos que todos los arcos contienen la siguiente información:

evento [if precondición] [/evento_disparado [if precond_disparo]]

Por tanto, se puede generar la declaración de las precondiciones de los eventos en Oasis a partir de toda la información anterior. En Oasis, los eventos que generan y destruyen instancias (*new, destroy*), tienen como precondiciones por defecto la no existencia/existencia respectivamente, de la instancia considerada. Asimismo, cualquier otro evento asume como precondición por defecto la existencia del objeto participante.

A partir de la figura 5.7, podemos definir la precondición asociada al evento *ocupar* de la clase *habitación* en C-Oasis como:

ocupar(p,h,t) if disponible(h,true,t)

En F-Oasis se define análogamente como:

T=T1'ocupar(p,h,t) if disponible(h)=true

Y, finalmente, en L-Oasis tendríamos la siguiente precondición:

ocupar(p,h,t) if disponible(h,t)=true

Relaciones de disparo

Como acabamos de ver, la información de los eventos que pueden ser disparados cuando una transición de estado se produce está contenida en el arco que representa al evento causante de tal transición. De acuerdo con la sintaxis con que se especifican en Oasis las relaciones de disparo, se genera la siguiente declaración:

object::evento_disparado if condición-de-disparo

donde:

* **object** es el objeto destinatario.
* **evento_disparado** es el evento cuya ocurrencia se activa automáticamente como consecuencia de la ocurrencia de **evento.**
* **condición de disparo** es la condición que debe ser satisfecha para que el evento indicado sea disparado. En ella se incluye el evento inductor del disparo, que es el que origina el cambio de estado etiquetado en la transición analizada.

El diseñador puede manipular la granularidad del objeto destinatario de acuerdo con las posibilidades facilitadas por Oasis. También puede añadir en este contexto nuevas relaciones de disparo en las que un evento se active como consecuencia de la satisfacción de cierta condición, haciendo uso de la flexibilidad y capacidad expresiva de Oasis.

En definitiva, la transición de los productos resultantes del AOO a una especificación en Oasis se sistematiza hasta tal punto que se posibilita la realización de tal proceso con soporte automático. Un compilador de alto nivel acepta como entrada los modelos gráficos producidos en la fase de AOO (un DCC y los DEC correspondientes por cada clase) e interacciona con el diseñador mostrándole en cada paso las diferentes alternativas de diseño que se presentan y esperando una

respuesta para continuar el proceso, hasta generar la especificación en Oasis equivalente a los modelos gráficos iniciales.

En [60,78,97,96,122], se presentan diferentes ejemplos referentes a casos prácticos resueltos utilizando OO-Method. Se incluyen en todos ellos los modelos gráficos resultantes del AOO y las especificaciones generadas en Oasis.

Se hace realidad en este entorno de trabajo uno de los objetivos de esta Tesis consistente en el diseño y desarrollo de entornos avanzados de Software que hagan operacional el Paradigma de la Programación Automática dentro de un entorno formal y OO bien definido.

Pero ésto no es todo. ¿Qué ocurre con la implementación? El objetivo final de todo proceso de producción de Software es, evidentemente producir Software de calidad. Hemos llegado hasta la fase de Diseño, y disponemos de una especificación en Oasis, generada automáticamente a partir de los resultados del Análisis. Pero, ¿qué partido le vamos a sacar este Diseño?. ¿Cómo vamos a llegar al producto final?. La respuesta OO-Method va a seguir siendo la misma:

> *automáticamente a partir del producto del Diseño (la especificación en Oasis), vamos a obtener un producto Software CORRESPONDIENTE a la Especificación.*

5.4 Implementación

Se ha visto en el capítulo 5 que una especificación en Oasis se corresponde con una Teoría de Primer Orden, clausal en el caso de C-Oasis, ecuacional en su versión funcional F-Oasis y clausal con igualdad en el caso de L-Oasis.

Como la Programación Lógica hace representables y manipulables en un ordenador tales Teorías a través de Lenguajes de Programación bien conocidos como Prolog [13] en el campo clausal, o Axis [29], Rap [66], etc en el funcional, se pueden *compilar* las especificaciones en Oasis generando sus programas lógicos equivalentes. La animación de tales programas va a constituir un producto Software equivalente a la especificación inicial, con lo que se proporciona un entorno de producción automática de Software en un marco OO y declarativo, como se muestra en la figura 5.9. A partir de los modelos resultantes de la fase de Análisis

se genera, con la ayuda de un traductor interactivo, la especificación en Oasis equivalente, y nuevamente de forma automática se generan a partir de dicha especificación los Programas Lógicos clausales, ecuacionales o clausales con igualdad correspondientes a la especificación. Detalles concretos de cómo se han construido todos esos traductores que componen el entorno de producción automática de Software pueden encontrarse en [25], [37], [40], [67], [109], [141] . En todos estos trabajos, no sólo se genera el programa lógico equivalente a una especificación en Oasis, sino que también se construyen prototipadores o animadores de esos programas.

La implementación final proporciona en una estación de trabajo un conjunto de menús con los que los agentes autorizados activan eventos. Si estos eventos son relevantes (es decir, si sus precondiciones se satisfacen y en el nuevo estado al que llevan al objeto(s) afectado(s) las restricciones de integridad definidas en sus clases se satisfacen), se añaden a la vida del Sistema. En caso contrario, el evento es rechazado.

En paralelo, la actividad de la Sociedad de Objetos se traduce en que a cada objeto se le comprueban sus condiciones de disparo para activar el evento adecuado cuando alguna de esas condiciones se cumpla.

En cualquier momento, usuarios autorizados pueden consultar el estado de cualquier objeto. Estas observaciones pueden ser tan triviales como conocer los valores actuales de los atributos de un objeto seleccionado (estado actual del objeto), u observaciones en cualquier instante anterior, o mucho más complicadas si se utiliza como potente lenguaje de consulta el lenguaje de primer orden subyacente a la Teoría de Primer Orden equivalente a la Especificación (como vimos en el capítulo 5).

En cualquier caso, cualquier consulta se evalua sobre las trazas de los objetos (sus vidas) para devolver la información requerida.

En [141] a partir de una versión de C-Oasis que hace uso de una expresividad lógica completa de primer orden, se genera la definición de una Base de Datos Deductiva implementada en el DSIC ([26]), a través de un traductor que toma como fuente una especificación en R-Oasis, y produce automáticamente un programa en BIM-Prolog. Este programa constituye una definición de BDD tal y como se define en [26].

La animación de la BDD se realiza incluyendo como nuevos hechos de la BDD los eventos relevantes, y en cualquier instante se pueden

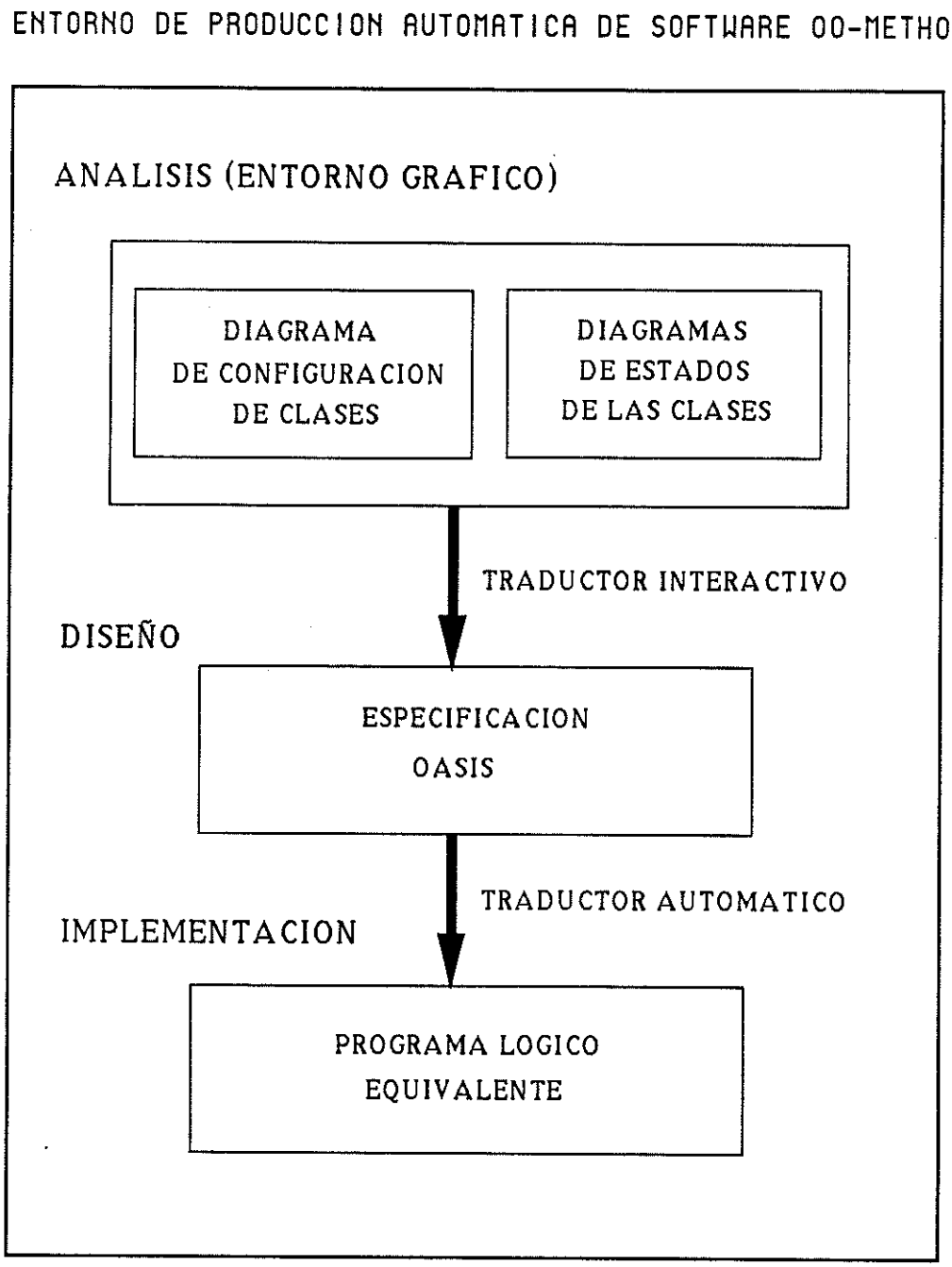


Figura 5.9: Representación del entorno de producción automática de Software OO-Method. A partir de los modelos gráficos obtenidos en la fase de Análisis , un traductor de alto nivel genera una especificación en Oasis que constituye el resultado del Diseño. A partir de ella, se generan automáticamente los programas lógicos clausales o ecuacionales que son equivalentes a la especificación

efectuar observaciones para conocer el estado de los objetos de la BD. El sistema implementado es pasivo.

En [25], se implementa un Sistema Activo a través de un mecanismo cercano al propuesto en Polka ([38]). Una especificación en R-Oasis es traducida nuevamente a su programa equivalente en Prolog, incluyendo las relaciones de disparo.

La animación de la especificación incluye un modo de funcionamiento automático en el que las condiciones de disparo se están comprobando continuamente para disparar aquellos eventos cuya condición de disparo se satisfaga.

En [67] se implementa un traductor de F-Oasis a Axis. La especificación es convertida en una presentación algebraica equivalente, cuya animación se lleva a cabo representando el estado del Sistema por medio de un término del sort que lo representa.

En esta traducción, atributos, eventos y precondiciones se representan como operadores (la única expresividad de que se dispone en una presentación algebraica)

El trabajo presentado en [109] implementa un traductor análogo al anterior, pero que utiliza RAP como lenguaje objeto. Como RAP da soporte al uso de variables lógicas, el tipo de consultas que se pueden realizar es más rico.

En [37] se implementa un traductor de especificaciones en L-Oasis a Europa, generando automáticamente un entorno de prototipación lógico-ecuacional a través de un animador o ejecutador de las especificaciones. El trabajo de [40] completa el entorno anterior, permitiendo utilizar cualquiera de los operador entre clases propuestos en Oasis.

En resumen, se dispone de un avanzado entorno formal de producción de Software con un alto grado de automatización, compuesto por un conjunto de traductores de alto nivel que, a partir de una especificación gráfica OO del SO problema, generan una implementación equivalente a esa especificación en distintos Lenguajes de Programación Lógica.

Capítulo 6

Conclusiones

Los nuevos vientos que soplan en el campo de la Ingeniería de Software pocas veces rebasan el límite de las buenas intenciones. La puesta en práctica de un nuevo paradigma basado en la noción de programación automática y haciendo uso de modelos descriptivos cercanos al mundo real y métodos formales, tropieza tradicionalmente con un 'pero' archiconocido:"Eso está muy bien. Cuando se disponga de tales métodos, ya hablaremos".

Un entorno de producción automática de Software basado en 'tales métodos' ha sido diseñado y desarrollado en esta Tesis, con el fin de disponer de herramientas operacionales basadas en ese nuevo Paradigma con el que el proceso de producción de Software tiene como piezas básicas:

1) la generación automática de una especificación que denote una teoría que tenga un modelo naturalmente isomorfo al dominio del Sistema considerado.

2) la transformación de esa especificación en una teoría formulada en un lenguaje de programación, preservando el modelo anterior.

El trabajo presentado se sitúa dentro del intenso esfuerzo de investigación realizado en la actualidad por cientos de investigadores en la formalización del modelo OO como modelo idóneo para obtener especificaciones de Sistemas del mundo real, y el estudio de nuevos métodos formales que, en un contexto OO bien definido, suministre una Metodología operacional de producción de Software de calidad.

Los objetivos alcanzados han sido los siguientes:

* Se ha definido intuitiva y formalmente un modelo OO y Ontológico O^3 que, partiendo de conceptos básicos en Sistemas de Información enriquecidos con nociones ontológicas, constituye un marco de referencia elegante y potente para describir Sistemas del mundo real.

  La formalización se ha efectuado en un marco algebraico que da cuenta de las nociones básicas del O^3 de una forma clara y simple. La introducción de los operadores de clase en ese contexto algebraico es una aportación que hace posible introducir un mecanismo constructivo en la especificación de una Sociedad de Objetos Complejos.

* Se ha diseñado y desarrollado un Lenguaje de Especificación OO llamado Oasis, que es la herramienta usada para especificar Sistemas de acuerdo con los principios del modelo O^3.

  La generación de las Teorías de Primer Orden equivalentes a una especificación Oasis permite asociar al lenguaje una semántica declarativa y operacional precisas, lo que posibilita la creación de un entorno de Prototipación Rápida en el que los usuarios pueden validar una especificación a través de su ejecución.

* Se ha diseñado y desarrollado una Metodología llamada OO-Method, que proporciona un entorno operacional de producción de Software basado en tres pilares fundamentales:

  1) Es un entorno fiel al Paradigma de Programación Automática en el sentido que la transición de Análisis a Diseño, y de Diseño a Implementación se efectúan de una manera automatizada.

     Se generan dos tipos de modelos gráficos resultantes de la fase de AOO en un entorno gráfico amigable. Estos modelos gráficos constituyen la entrada de un traductor interactivo de alto nivel que genera **automáticamente una especificación Oasis equivalente.**[1] Esta especificación es a su vez la entrada de un compilador de alto nivel que produce de forma **automática los Programas Lógicos equivalentes.**

  2) Uso de métodos formales de especificación, que dan sentido a la realización de procesos de validación (mediante prototipación automática de una especificación Oasis) y de auditoría (que con el uso de métodos fieles al Paradigma de Programación Automática se reduce a verificar que un proceso de traducción automática se ha realizado correctamente). Con estos métodos, se puede hablar de nociones como completitud, consistencia,.. con propiedad, y no de la

[1] Este traductor está siendo implementado en dos fases. El conversor de un DCC en la base de una especificación Oasis ya ha sido realizado [108]. El traductor de DEC's que completa esa especificación base está actualmente en desarrollo.

forma arbitraria en que se utilizan tales términos en la jerga del CASE tradicional.

3) Utilización del modelo objetual O^3 formalmente definido, que subyace a toda la Metodología, con la propiedad esencial que permite describir la realidad de una forma natural, reduciendo el 'gap' semántico entre el "cómo es percibido" el Sistema y el "cómo es representado" en el producto Software finalmente obtenido.

El trabajo se enmarca dentro de los proyectos CICYT de investigación SINTESIS[2] y su continuación PROTESIS. La Metodología está siendo aplicada al Diseño y Desarrollo de Sistemas Reales Complejos en diferentes entornos (Gestión de Expedientes de diversos tipos en distintos Organismos Públicos, Diseño y Desarrollo de un Sistema de Gestión de Impresoras para la TVV (Radio Televisión Valenciana), Desarrollo de una Gestión de Personal,...), y sus resultados están siendo validados en laboratorios de investigación y desarrollo de Software internacionales (HP Labs., Bristol (UK)), y usados en empresas españolas (DRAC, CLIA) en el marco de un proyecto de Enseñanza Cooperativa de Ingeniería del Software desde 1989.

Las tareas en las que se continúa trabajando intensamente se pueden agrupar en los siguientes puntos:

* Enriquecimiento de la expresividad del Lenguaje de Especificación Oasis y de su formalización algebraica subyacente con:

  1) Declaración de Restricciones de Integridad Dinámicas, que establezcan relaciones entre diferentes estados. La captación de estos aspectos dinámicos obliga a realizar ampliaciones modales en las Lógicas utilizadas. Un monitor de Restricciones de Integridad Dinámicas ha sido ya implementado para la versión funcional pasiva de Oasis, y en esa linea el trabajo continua para introducir esas ampliaciones modales en la formalización del entorno, y para implementar tales monitores en todas las versiones de Oasis.

[2]Proyecto CYCIT 101.88.12.781.00 (Plan Nacional de I+D; Ayudas de los Programas Nacionales Científico-Tecnológicos y de Ciencias Sociales y Humanas)

2) Introducción y formalización de la noción de **agente** en la Especificación. Un agente es un activador potencial de eventos. La información de cuáles son los agentes asociados a cada evento se recoge en el Diagrama de Configuración de Clases, pero no se utiliza en fase de Diseño. La animación del Programa Lógico equivalente a una especificación Oasis puede enriquecerse si el Sistema identifica qué agentes son válidos, y qué eventos pueden ser activados por cada uno de ellos.

3) Introducir la noción de **proceso**, para inducir un cierto grado de estructura en los eventos que componen las trazas de los objetos de una clase. De esta forma un objeto se define como un proceso caracterizado por una expresión del álgebra de procesos utilizada.

   Con la ayuda de algunos operadores clásicos en el contexto de las álgebras de procesos, sería muy interesante poder expresar, por ejemplo, que:

$$P \rightarrow acP + bdP$$

   donde '+' denota elección y se interpreta como que las trazas correctas de las instancias de una clase P se componen de secuencias de un evento a más otro c, o de un evento b más otro d.

   Esto puede ser expresado en el estado actual de Oasis por medio de precondiciones, pero con una complejidad mayor.

4) Desarrollo de expresividades alternativas para el tiempo. En las versiones actuales, el tiempo se representa a través del tipo de los naturales, y todos los eventos y atributos vienen etiquetados con el valor del número natural que representa el instante temporal en el evento ocurre o el atributo es observado.

   Otras representaciones se están investigando y en particular la propuesta TSOS ([8]) basada en una lógica de intervalos temporales ha sido introducida de forma experimental en la versión clausal de Oasis ([127]).

* Desarrollar un marco formal algebraico sencillo y potente que permita formalizar un entorno de especificación Oasis en un contexto estrictamente algebraico, de forma que se pueda aplicar a la formalización de cualquier entorno de especificación caracterizado por un lenguaje de especificación dado.

  En nuestro caso, la idea subyacente es que toda especificación Oasis sea vista como un término con variables $t(x)$ perteneciente al lenguaje de términos con variables $T_\Sigma(x)$ engendrado por una cierta signatura Σ. Un lenguaje de especificación va a llevar asociada una signatura que lo caracteriza. Precisada esta signatura, las especificaciones escritas con el lenguaje serán términos con variables $t(x) \in T_\Sigma(x)$.

  Si se determina esta signatura $\Sigma = (S, \Omega)$, se tiene el medio constructivo para dar cuenta de estas especificaciones a través de una presentación algebraica. Además, haciendo uso de un lenguaje de programación funcional podemos disponer de implementaciones de tales especificaciones sin necesidad de construir traductores.

  Si en este contexto se formalizan los operadores entre clases propios del estado actual de Oasis, la introducción de nuevos operadores en el lenguaje sólo implica su definición en la presentación asociada al Lenguaje de Especificación. Así, la flexibilidad del entorno es completa porque un diseñador puede enriquecerlo especificando formalmente aquellos nuevos operadores que necesite.

* La amplia divulgación de los LPOO (C++,Smalltalk,Eiffel,...) y los rendimientos no demasiado satisfactorios en la actualidad de los Entornos de Programación Lógica, hace necesario el estudiar como generar código en tales lenguajes a partir de una especificación Oasis. El hecho de continuar trabajando en un entorno OO hace factible tal generación (básicamente, convirtiendo Clases y atributos asociados de Oasis en Clases con sus atributos en C++, y definiendo métodos en C++ para cada evento Oasis).

  El aspecto negativo reside en que se pierden las propiedades formales que se tienen dentro de un entorno lógico, porque no hay forma de generar EL programa C++ *equivalente* a la especificación Oasis de forma automática. De todas formas, sí que se

puede generar un conjunto de programas a partir de la especificación que constituyan una maqueta de lo que finalmente será el producto Software definitivo. Esto conecta el trabajo presentado en esta Tesis con todos los productos Software relacionados con LPOO y BDOO que empiezan a invadir el mercado en la actualidad, y supone el disponer de un práctico estado intermedio entre las metodologías clásicas de producción de Software que se vienen utilizando hasta nuestros días, y el nuevo Paradigma de Programación Automática basada en un modelo OOO del que OO-Method pretende ser prototipo.

* Elaboración de una herramienta CASE que de soporte a OO-Method como Metodología. La práctica totalidad de los métodos implicados ya han sido implementados, por lo que el esfuerzo ahora se centra en generar un entorno común que los englobe en una estación de trabajo lógica única, con una perspectiva más comercial.

* Capturar aspectos de paralelismo en el Lenguaje de Especificación para reflejar la naturaleza intrínsecamente paralela de los SI.

  En este contexto, los Lenguajes de Programación Paralela OO como POOL ([5]) constituyen una herramienta de implementación especialmente apropiada para manipular las relaciones de disparo de los objetos de una clase como actividad propia de los cuerpos ('body') definidos en la clase correspondiente.

Capítulo 7

Bibliografía

Bibliografía

[1] Ackoff,R.L. *Towards a System of System Concepts* Management Science, Vol.17, July 1971.

[2] ACM-SIGSOFT *Rapid Prototyping Workshop* ACM Sofw.Eng.Not.1982,7,(5)

[3] Agresti,W.W. *New Paradigms for Software Engineering* IEEE Computer Society Press,Washington 1986

[4] Alpuente.M.:Ramírez,M.J. *The Logic+Ecuational EUROPA environment and its application to the rapid prototyping of Data Base applications* in Proc of the 7th IASTED Int. Conf. on Expert Systems, theory and applications, Los Angeles, USA, 12-15 Dec.1990

[5] America,P.
POOL-T: A Parallel Object-Oriented Language in Object-Oriented Concurrent Programming, Yonezawa,A.:Tokoro,M. (eds.), pp.199-220, The MIT Press,Cambridge,MA,1987.

[6] Atkinson et.al. *The Object-Oriented Database System Manifesto* 1st Int.Conference on Deductive and Object-Oriented Databases, Kim,W.et al.(eds.), 1989, pp.40-57

[7] Balzer,R:Cheatman,T.E.:Green,C.
Software Technology in the 1990's: Using a New Paradigm IEEE Computer,Nov.1983,16,(11).pp.3-16

[8] Barbic,F.:Maiochi,R.:Pernici,B. *Automatic Deduction of Temporal Information* Universidad Politécnica de Milano, Research Report, 1990.

[9] Bear,S.:Allen,P.:Coleman,D.:Hayes,F: *Graphical Specification of object oriented systems* In Proceedings of Object-oriented Programming: Systems, Languages and Applications,1990

[10] Bergstra,J.A.:Klop,J.W. *Process Algebra for Synchronous Communication* Information and Control 60, pp.109-137, 1984

[11] Bergstra,J.A.:Klop,J.W. *Algebra of Communicating Processes with Abstraction* Theoretical Computer Science 37, pp.77-121, 1985

[12] Bergstra,J.A.:Klop,J.W. *Algebra of Communicating Processes* In deBakker and al. (eds.) Mathematics and Computer Science (CWI Monographs 1), pp.89-138, North-Holland 1986.

[13] *BIM Prolog Reference Manual*,ISS,Belgium,1990

[14] Boehm,B.W. *A spiral model of software development and enhancement* IEEE Computer,1988,21,(5),pp.61-72

[15] Booch,G. *Object Oriented Design with applications* Benjamin Cummings Publishing Company,Inc 1991

[16] Boulding,K. *General System Theory - The Skeleton of Science* Management Science Vol.2, April 1956.

[17] Bouzeghoub,M:Metais,E. *Semantic Modelling of Object Oriented Databases* In Proc. of the Very Large Databases, Barcelona, September 1991, pp.3-14.

[18] Bowen,K.A.,Kowalski,R.A. *Amalgamating language and metalanguage in logic programming* In Clark,K.L.,Tarnlund,S-A. (eds.). Logic Programming (pp.153-172),Academic Press,1982

[19] Brodie.M.L. *On Modelling Behavioural Semantics of Data* In Proc. of the 7th Int. Conference on Very Large Data Bases, Cannes, France, Sept.1981.

[20] Bunge,M. *Treatise on Basic Philosophy:Vol.3:Ontology I:The Furniture of the World* Reidel,Boston 1977.

[21] Bunge,M. *Treatise on Basic Philosophy:Vol.4:Ontology II:A World of Systems* Reidel,Boston 1977.

[22] Bubenko,J.A.:Olive,A. *Dynamic or Temporal Modelling? An Illustrative Comparison* SYSLAB Working Paper 117,Nov.1986

[23] Canós,J.H.:Pastor,O. *Object Oriented and Functional Specification of Information Systems* in Proc. of DEXA-91,Berlin,1991

[24] Canós,J.H.:Pastor,O. *Adding Logic Variables to a Functional and Object Oriented Specification Language* in Proc. of IASTED Conference on Applied Informatics,Zurich,1991.

[25] Casado,V. *Oasis: Especificación de Sistemas de Información Abiertos y Activos* Proyecto final de carrera dirigido por I.Ramos, Facultad de Informática (UPV) 1990.

[26] Casamayor,J.C.:Celma,M.:Mota,L.:Pastor,M.A.:Marqués,F *Deductive Data Base Management Systems: a Prototype* In Proc. of the IAESTED Conference on Applied Informatics, 1990

[27] Checkland,P. *Systems Thinking, Systems Practice* J.Wiley and Sons 1981.

[28] Coad,P.:Yourdon,E *Object-Oriented Analysis*, Yourdon Press 1990.

[29] Coleman,D.:Dollin,C.:Gallimore,R.:Arnold,P.:Rush,T: *An Introduction to the Axis Specification Language*, Technical Report,HP-Labs. Bristol(UK) 1988

[30] Coleman,D.:Hayes,F. *Getting the best out of objects: Hewlett-Packard's experiences.* In Proceedings of Technology of Object-Oriented Languages and Systems Conference,1991.

[31] Coleman,D.:Hayes,F. *Coherent Models for Object-Oriented Analysis* HP-Labs.Technical Report HPL-91-47,1991

[32] Coleman,D.:Hayes,F.:Bear,S. *Introducing objectcharts or how to use statecharts in object oriented design* IEEE Transaction in Software Engineering (to appear)

[33] Constantine,L. *Panel in Structured Analysis and Object Oriented Analysis* in Proc. of the Conference on Object-Oriented Programming:Systems, Languages and Applications (OOPSLA-90), Ottawa(Canada), : ed. Meyrowitz,N.,ACM-Press 1990

[34] Costa,J-F.:Dionisio,F.:Sernadas,A.:Ehrich,H.D. *The Ultimate (Finitely) Cocomplete Category of Sequential Processes* Research Report, INESC 1990.

[35] Costa,J-F.:Sernadas,A. *The Inequational Logic of Liveness in Concurrent Systems* Research Report, INESC 1991.

[36] Costa,J-F.:Sernadas,A. *Process Models within a Categorial Framework* Research Report, INESC 1991.

[37] Cuevas,J *Generación Automática de un Entorno de Prototipación Lógico-Ecuacional, a partir de un Lenguaje de Especificación Orientado a Objetos (Oasis).* Proyecto Fin de Carrera dirigido por O.Pastor, Dic.1991, Facultad de Informática,UPV (Valencia).

[38] Davison,A. *Blackboard Systems in Polka* Dept.of Computing, Imperial College (London),1988.

[39] Dewar,R.:Grand,A.:Liu,S.:Schwartz,J *Programming by refinement, as exemplified by the SETL representation sublanguage* ACM Trans. Program. Lang. Sys., 1979,1, (1), pp.27-49.

[40] Devesa,J. *Implementación de Operadores entre Clases en un Entorno de Especificación de Sistemas de Información Orientados a Objeto* Proyecto Fin de Carrera dirigido por Oscar Pastor, 1992, Facultad de Informática, U.P.V. (Valencia)

[41] Ehrich,H.D.: Goguen,J.A.: Sernadas,A. *A Categorial Theory of Objects as Observed Processes* In: deBakker,J.: deRoever,W: Rozenberg,G. (eds.).Foundations of Object Oriented Languages. (Proc.REX School/Workshop), Noordwijkerhood(NL),1990.LNCS 489,Springer-Verlag,Berlin 1991, pp.203-228.

[42] Ehrich,H.D.:Sernadas,A. *Algebraic Implementation of Objects over Objects* In: deRoever,W.(ed.) Stepwise Refinement of Distributed Systems:Models, Formalisms, Correctness (Proc.REX'90) , Mood(NL),1990. LNCS 430, Springer Verlag, Berlin, 1990, pp. 239-266

[43] Ehrich,H.D.:Sernadas,A.:Sernadas,C *From Data Types to Object Types* Journal of Information Processing and Cybernetics, EIK 26 (1990) 1/2,33-48.

[44] Ehrich,H.D.:Sernadas,A *Fundamental Objects Concepts and Constructions* Proc. of the Second International IS-CORE Workshop. Imperial College,London-1991, pp.1-24.

[45] Ferrandis,I:Ramos,I.:Canos,J. *Herramientas Gráficas de Prototipación Automática* Research Report, DSIC 1991.

[46] Fiadeiro,J.:Maibaum,T. *Towards Object Calculi* Proc. of the Second International IS-CORE Workshop. Imperial College,London-1991, pp.129-178.

[47] Fishman,D.H., Annevelink,J., Becch,D, Chow,E., Connors,T., Davis,J.W., Hasan,W., Hoch,C.G., Kent,W., Leichner,S., Lyngbaek,P. ,Mahbod,B., Neimat,M.A., Risch,T., Shan,M.C., Wilkinson,W.K. *Overview of the Iris DBMS* In, Kim,W. ,Lochovsky,F.H. (eds.) Object Oriented Concepts, Databases and Applications, ACM-Press 1989, pp. 219-250

[48] Gane,C.:Sarson,T. *Structured System Analysis* Englewood Cliffs, NJ Prentice Hall 1979.

[49] Gehani,N.:Jagadish,H.V. *Ode as an Active Database:Constraints and Triggers* Proceedings of the 17th International Conference on Very Large Data Bases,VLDB 1991, Barcelona.

[50] Goguen,J. *Rapid Prototyping in the OBJ executable specification language* ACM Software Engineering Not. 1982,7 (5), pp.75-84.

[51] Goguen,J. *Categorical Foundations for General Systems Theory* Advances in Cybernetics and Systems Research, Transcripta Books, 1973, 121-130.

[52] Goguen,J. *Objects* International Journal of General Systems,1 (1975), 237-243.

[53] Goguen,J. *A Cathegorical Manifesto* Technical Report PRG-72, Programming Research Group, Oxford University, Marzo 1989. (to appear in Mathematical Structures in Computer Science)

[54] Goguen,J.:Messeguer,J *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics* Research Directions in OO Programming, MIT Press 1987.

[55] Goguen,J.:Thatcher,J.:Wagner,E. *An Initial Algebraic Approach to the Specification, Correctness, and Implementation of Abstract Data Types* in R.Yeh (ed) Current Trend in Programming Methodology, Vol.4, Prentive-Hall 1978,80-149

[56] Goldberg,A:Robson,D. *Smalltalk 80: The Language and its Implementation* Addison-Wesley,New York 1983.

[57] González del Río,A. *MOT: Modelo Orientado a Transacciones* Research Report, DSIC-UPV (Valencia, 1991).

[58] Hammer,M.:McLeod,D. *Database Descriptions with SDM: A Semantic Database Model* ACM Transactions on Database Systems, vol.6, n.3, pp. 351-386, Sept.1981.

[59] Harel,D. *Statecharts:A visual formalism for complex systems* Science of Computer Programming, 8:231-274,1987

[60] Hernández Llorens, Jose Miguel. *Conversión de un Entorno de Análisis y Diseño en MOT a un Entorno de Prototipación Orientado a Objetos OASIS.* Proyecto Final de Carrera, dirigido por O.Pastor. Facultad de Informática (UPV). Valencia, 1991

[61] Hill,P.M., Lloyd,J.W. *Metaprogramming for Dynamic Knowledge Bases* Technical Report CS-88-18, Dept.of Computer Science,University of Bristol(England),1988.

[62] Hill,P.M., Lloyd,J.W. *Analysis of Metaprograms* In H.D.Abramson, M.H.Rogers (eds.), Metaprogramming in Logic Programming, pages 417-434, MIT Press,1989. (also in Proc. of the Meta88 Workshop, June 1988)

[63] Hoare,C.A.R. *Communicating Sequential Processes* Prentice Hall International,1985

[64] Van Horebeek,I. *Algebraic Specifications:From many-sorted algebras to a practical Specification Language*

[65] Hull,R.,King,R. *Semantic Database Modelling* ACM Computing Surveys,Vol.19,N.3,Sept.1987.

[66] Hussmann,H. *Rapid Prototyping for Algebraic Specification, RAP System Users's Manual* Version 2.0, Report Universitat Passau,Fakultat fur Mathematik und Informatik,July 1990.

[67] Iborra,Ana *Introducción de Herencia en un Entorno de Prototipación OO y Funcional* Proyecto Final de Carrera, dirigido por Oscar Pastor. Facultad de Informática, U.P.V., 1991

[68] IS-CORE 91 IS-CORE: Information Systems - Correctness and Reusability Esprit-2 BRA WG 3023, Second Year Report, Septiembre 1991.

[69] Jungclaus,R.:Hartmann,T.:Saake,G. *Languages Features for Object Oriented Conceptual Modelling* In: Teory,T.(ed.) Proc. 10th Int.Conf. on the ER-Approach,San Mateo(CA),1991.

[70] Jungclaus,R.:Hartmann,T.:Saake,G.:Sernadas,C. *Introduction to Troll.- A Language for Object Oriented Specification of Information System* Second International IS-CORE Workshop. London-1991.

[71] Jungclaus,R.:Saake,G. *Activity Specification in a Framework for Formal Object-Oriented System Development* Research Report, TUBS 1991.

[72] Jungclaus,R.:Saake,G.:Sernadas,C. *Formal Specificacion of Object Systems* In: Abramsky,S.;Maibaum,T. (eds.) Proc. TAPSOFT'91,Brighton(UK), 1991. LNCS 494,Springer-Verlag,Berlin pp.60-82

[73] Kim,W.:Ballou,Nat:Chou,H.T.:Garza,J.F. *Features of the ORION Object-Oriented Database System* In: Kim,W.:Lochovsky,F.H. (eds.) Object Oriented Concepts, Databases and Applications, ACM-Press 1989, pp.251-282

[74] Kim,W. *Object-Oriented Databases: Definition and Research Directions* IEEE Transacions on Knowdledge and Data Engineering 2 (1990) pp.327-341

[75] Kowalski,R. *Logic as a Database Language* Computing Surveys, 1984.

[76] Lindgreen,P. (ed.) *A Framework of Information System Concepts* Interim Report,IFIP WG 8.1 (FRISCO), Mayo 1990

[77] Lloyd,J.W. *Foundations of Logic Programming* Second,Extended Version Springer Verlag, 1987

[78] López Belda,L. *Análisis, Diseño e Implementación en Oasis de un Gestor de Impresoras para la RTVV* Proyecto Final de Carrera, dirigido por O.Pastor, Facultad de Informática, UPV, 1990

[79] Loux,M.J. *Substance and Attribute.A Study in Ontology.*Reidel,1978

[80] deMarco,T. *Structured Analysis and System Specification* Englewood Cliffs, NJ., Prentice Hall, 1979.

[81] Meyer,B. *Object Oriented Software Construction* Prentice-Hall, Englewood Cliffs 1988.

[82] Mellor,S.J.:Shlaer,S. *Object Oriented System Analysis: Modelling the world in data.* Prentice Hall,1988

[83] Mendelson,E. *Introduction to Mathematical Logic* 2nd Edition, Van Nostrand, Princeton,N.J.,1979

[84] Meyer,B. *Object Oriented Software Construction* Prentice-Hall 1988.

[85] Middeldorp,A. *Counterexamples to the completeness results of basic narrowing* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, 1991.

[86] Milner,R. *A Calculus of Communicating Systems* Lecture Notes in Computer Science,vol 92,Springer-Verlag,1980

[87] Nelson,M.L. *An Object Oriented Tower of Babel* OOPS Messenguer, ACM Press, Volume 2 number 3, Julio 1991.

[88] Nierstratz,O.M. *What is an Object in Object Oriented Programming?* in Objects and Things, ed. D.Tsichritzis,Universite de Geneve,Centre Universitaire d'Informatique 1987

[89] Paris Urcaregui, J. *Tratamiento de la dinámica interobjetual mediante leyes* Proyecto Final de Carrera, dirigido por I.Ramos, Facultad de Informática, UPV 1991.

[90] Pastor,O.:Hayes,F.:Bear,S. *OASIS: An Object Oriented Specification Language* Research Report, DSIC,1991. Aceptado para presentación en el CAiSE-92, Manchester (UK)

[91] Pastor,O.:Ramos,I:,Canos,J.H. *Object Oriented and Relational Specification Of Information Systems* In the Proc. of the 2nd International Workshop of the Deductive Approach for DB and IS, Aiguablava (Catalonia), 1991

[92] Philips,J. *Self-described programming environments: an application of a theory of design to programming systems* PhD Thesis, Standard University, USA, 1982

[93] Ramos,I. *Logic and OO Databases: a Declarative Approach* Proc. of the DEXA 90, Springer-Verlag 1990

[94] Ramos,I.,Canos,J.H.,Forradellas,J.,Oliver,J. *A Conceptual Schema Specification System for Rapid Prototyping* Proc.of the XI IASTED Conference on Applied Informatics,Insbruck,Feb.1990

[95] Ramos,I:,Pastor,O.:Casado,V. *OO and Active Formal Information System Specification* In Proc, of DEXA-91, Springer-Verlag,Berlin,1991

[96] Romero Sánchez,J. *Análisis y Diseño del Sistema de Gestión Presupuestaria de la Generalitat Valenciana usando Oasis como herramienta de Diseño* Proyecto Final de Carrera, dirigido por Oscar Pastor, Facultad de Informática, U.P.V. 1992.

[97] Romeu Iborra,C. *OO-METHOD: Análisis y Diseño basado en el Paradigma de Programación Automática* Proyecto Final de Carrera dirigido por O.Pastor. Facultad de Informática (UPV), 1991

[98] Rumbaugh,J.:Blaha,M.:Premerlani,W.:Eddy,F.:Lorensen,W. *Object-Oriented Modelling and Design* Prentice Hall 1991.

[99] Rush,T.:Harry,P.:Ferguson,T.:Oliver,H. *Case studies in HP-SL* Technical Report,HPL-90-137, HP-Labs (Bristol,1990)

[100] Ryan,M.:Goldsack,S.:Cunnigham,J.:Fiadeiro,J.:Quirk,B.:Booth,J. *Structure in MAL* Marzo,1990 New Forest Report NFR/WP1.1/IC/M/001/B.

[101] Ryan,M. *Structure in MAL-2* Julio,1990 New Forest Report NFR/WP1.1/IC/M/001/A.

[102] Saake,G:Jungclaus,R *Konzeptionelle Modellierung von Objecktgesellschaften* In: Appelrath,H.J. (ed.) Proc. Datenbanksysteme fur Buro, Technik und Wissenschaft BTW'91, Kaiserlautern,1991. IFB 270, Springer-Verlag, Berlin,1991,pp. 327-343

[103] Saake,G.:Jungclaus,R. *Information about Objects versus Derived Objects* In Proc. Workshop on Foundations of Models and Languages for Data and Objects. Goeehrs,J.:Heuer,A. (eds.), Aigen,Informatik-Bericht 90/3, TU Clausthal, 1990, pp.59-69.

[104] Saake,G. *Conceptual Modelling of Database Application* In Proc. Information Systems and Artificial Intelligent: Integration Aspects. D.Karagiannis (ed.), Ulm 1990, LNCS 474, Springer-Verlag 1991 pp.213-232

[105] Saake,G. *Descriptive Specification of Database Object Behaviour* Data and Knowledge Engineering , North Holland, Vol.6, N.1,1991, pp. 47-73

[106] Saake,G.:Jungclaus,R. *Specifications of Database Dynamics in the Oblog+ Language* Research Report, TUBS 1991.

[107] Saake,G:Jungclaus,R *Specification of Database Applications in the Troll Language* In: Harper,D.(ed.) Proc.Int.Workshop on the Specification of Database Systems,Glasgow,1991.Springer-Verlag,London 1991.

[108] Sanz,S. *Desarrollo de una interface Gráfica para Oasis* Proyecto final de carrera dirigido por O.Pastor, Facultad de Informatica (UPV), 1991.

[109] Segura,J.*Traductor de FMOL (Lenguaje de Especificación de Sistemas de Información Basado en el Modelo OO) a RAP (Lenguaje Ecuacional con Soporte Para Variables Lógicas)* Proyecto final de carrera dirigido por O.Pastor, Facultad de Informatica (UPV), 1990.

[110] Sernadas,A.:Ehrich,H.-D. *What is an object, after all?* Proc. IFIP Working Conference DS-4, Meersmann,R.; Kent,W. (eds.) North-Holland, Amsterdam 1991

[111] Sernadas,A.:Fiadeiro,J:Sernadas,C.:Ehrich H-.D. *The Basic Building Blocks of Information Systems* Proc. IFIP 8.1 Working Conference, Falkenberg,E., Lindgreen,P (eds.), North Holland, Amsterdam,1989, 225-246

[112] Sernadas,C.: Gouveia,P.: Silva,L.: Lopes,A. *Objects as Structuring Units for Incorporating Dynamics in Deductive Conceptual Modelling* In Proc. of the Int. Workshop on the Deductive Approach to Information Systems and Databases, pp. 93-110, Catalonia 1990.

[113] Sernadas,C.:Gouveia,P.:Saake,G.:Jungclaus,R. *From Deductive to Object Oriented Database Descriptions* Research Report, INESC 1991.

[114] Sernadas,C.: Gouveia,P.: Gouveia,J.: Sernadas,A.: Resende,P. *The Reification Dimension in Object-Oriented Data Base Design* In Proc. International Workshop on Specification of Data Base Systems. Glasgow, Scotland, 3-5 July 1991.

[115] Sernadas,C.: Resende,P.: Gouveia,P.: Sernadas,A. *In-the-large Object-Oriented Design of Information Systems* In Proc. IFIP 8.1 Working Conference on the Object Oriented Approach in Information Systems, Quebec City (Canada), 1991.

[116] Sernadas,A.:Sernadas,C:Ehrich,H.D. *Object Oriented Specification of Databases: An Algebraic Approach.* Proc. 13th Int.Conf. on Very Large Data Bases VLDB'87,Brighton,1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107-116.

[117] Sernadas,A.: Sernadas,C.: Gouveia,P.: Resende,P.: Gouveia,J.: Silva,L. *OBLOG: An Informal Introduction* Research Report, INESC 1991.

[118] Shoenfield,J. *Mathematical Logic* Addison-Wesley, Reading,Mass, 1967

[119] Shipman,D. *The Functional Data Model and the Data Language DAPLEX* In ACM Transactions on Database Systems, vol.6, n.1, pp.140-173, March 1981.

[120] Smith,M:Tockey,S. *An Integrated Approach to Software Requirements Definition Using Objects* Seattle,WA: Boeing Commercial Airplane Support Division.

[121] Stroustrup,B. *The C++ Programming Language* Addison Wesley, Reading,Mass 1986.

[122] Tarazón Muñoz,M.J. *Análisis y Diseño del Sistema de Gestión Tributaria de la G.V. con Oasis como herramienta de Diseño* Proyecto final de carrera dirigido por O.Pastor, Facultad de Informatica (UPV),1991.

[123] Taylor,T.:Standish,T.A. *Initial thoughts on rapid prototyping* ACM Softw. Eng. Not.,1982,7,(5),pp. 160-166

[124] deTroyer,O. *Schema Object Types: A New Approach to Modularization in Conceptual Modelling.* Research Report, Tilburg University, 1991.

[125] deTroyer,O. *The OO-Binary Relationship Model: A Truly Object-Oriented Conceptual Model.* In Proc. of the CAiSE 91 Conference, Mai 1991, Trondheim.

[126] Tsai,J.P.:Weigert,T. *HCLIE:a logic based requirements language for new software engineering paradigms* Software Engineering Journal, Julio 1981, pp. 137-151

[127] Tirado,P. *Introducción de TSOS para el tratamiento temporal en entornos Mol* Proyecto final de carrera dirigido por I.Ramos, Facultad de Informática (UPV),1991.

[128] Verheijen,G.:van Bekkum,J. *NIAM: An Information Analysis Method* In Proc. of IFIP TC-8 Conference on Comparative Review of Information Systems Methodologies (CRIS-1). Eds. Verrijn-Stuart,A.: Olle,T.W.: Sol, H.North-Holland, 1982.

[129] Verharen,E.M. *Object-Oriented System Development: An Overview.* Research Report, Tilburg University,1990.

[130] Wand,Y. *A Proposal for a Formal Model of Objects* In: Kim,W.: Lochovsky,F.H. (eds.) Object Oriented Concepts, Databases and Applications, ACM-Press 1989

[131] Wand,Y.:Weber,R. *An Ontological Model of an Information System* IEEE Transactions on Software Engineering, Vo.16, N.11, Nov.1990

[132] Wand,Y.:Woo,C.C. *An Approach to Formalizing Organizational Open Systems Concepts* ACM-Press, 1991 (ACM-0-89791-456-2/91/0010)

[133] Ward,P.:Mellor,S. *Structured Development for Real Time Systems* Englewood Cliffs, NJ Yourdon Press, 1985.

[134] Wegner,P. *Concepts and Paradigms of Object Oriented Programming* OOPS Messenguer, ACM Press, Volume 1 number 1, Agosto 1990.

[135] Weiser,S.P.:Lochovsky,F. *OZ+:An Object Oriented Database System* Object Oriented Concepts, Databases and Applications, ACM-Press 1989

[136] Wieringa,R. *Equational Specification of Dynamic Objects* in Proc. IFIP TC2 Working Conference on Database Semantics, Julio 1990.

[137] Wieringa,R. *A Conceptual Model Specification Language (CMSL version 2)* Technical Report,Dep.of Mathematics and Computer Science,Vrije Universiteit, Amsterdam,Abril 1991

[138] Wieringa,R. *Steps Towards a Method for the Formal Modeling of Dynamic Objets*, Data and Knowdledge Engineering,1991

[139] Yourdon,E. *Modern Structured Analysis* Englewood Cliffs, NJ. Prentice Hall,1989.

[140] Zave,P. *The operational versus the conventional approach to software development* Commun.ACM, 1984, 27,(2), pp.104-118

[141] Zuriaga,S. *Implementación de un Entorno de Producción de Software Orientado a Objetos y Deductivo.* Proyecto final de carrera dirigido por O.Pastor, Facultad de Informatica (UPV),1991.